Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Relatório final de pesquisa – Iniciação Científica

Algoritmos de Aproximação

Aluno:
Wesley Lima de Araujo

Supervisor:
Prof. Dra. Carla Negri Lintzmayer

Setembro/2020

Resumo

Em um problema de otimização combinatória desejamos encontrar uma melhor solução dentro de um domínio de soluções, de acordo com algum critério, sendo que as variáveis do problema possuem domínios discretos. Diversos problemas de otimização combinatória são $\mathcal{NP}$-difíceis, ou seja, não há esperança em encontrar algoritmos exatos eficientes para qualquer instância, a menos que $\mathcal{P} = \mathcal{NP}$. Nesse cenário, os algoritmos de aproximação são uma ferramenta útil, pois são executáveis em tempo polinomial e sempre devolvem uma solução cujo custo está garantidamente a um determinado fator de distância do custo da melhor solução. Este relatório fez parte de um projeto de iniciação científica que teve como objetivo introduzir o aluno à área de otimização combinatória através do estudo de algoritmos de aproximação. Nele são apresentados problemas de otimização combinatória, algoritmos de aproximação, resultados relacionados e conceitos de projeto de algoritmos estudados ao longo do projeto. Além disso, o relatório foi inteiramente escrito pelo aluno, o que faz com que o relatório em si seja um reflexo da compreensão do aluno sobre os conceitos estudados. Ao final do projeto, pôde-se concluir que o aluno desenvolveu uma boa noção sobre as áreas de otimização combinatória e projeto de algoritmos, fazendo com que o objetivo do projeto fosse atingido com sucesso.

Assuntos: Algoritmos de Aproximação; Otimização Combinatória.

Sumário

Introdução

A área de otimização combinatória engloba diversos problemas cujo objetivo é buscar uma solução ótima dentro de um conjunto de soluções viáveis para uma determinada entrada de um problema. Dependendo da função objetivo que define o custo de uma solução, uma solução ótima pode ser aquela de custo máximo ou mínimo, de forma que podemos estar falando de um problema de minimização ou de maximização.

Normalmente, a quantidade de soluções viáveis em um problema de otimização combinatória é gigantesca, mas sempre é finita. Assim, quando vamos resolver esse tipo de problema através de algoritmos, descartamos aqueles de força bruta devido à sua baixíssima eficiência computacional.

Muitos problemas de otimização combinatória são $\mathcal{NP}$-difíceis, fazendo com que não tenhamos esperança de encontrar algoritmos eficientes para resolvê-los, pois, isso só ocorreria se $\mathcal{P} = \mathcal{NP}$. Neste cenário, temos como uma abordagem para lidar com tais problemas os *algoritmos de aproximação*.

A ideia dos algoritmos de aproximação é sacrificar a otimalidade da solução em prol de uma melhor eficiência computacional, mas ainda garantindo que a solução devolvida esteja a algum fator de distância da solução ótima, qualquer que seja a entrada do problema. Por mais que isso pareça ser fácil, algumas vezes projetar algoritmos de aproximação eficientes é tão difícil quanto projetar algoritmos exatos eficientes.

Assim, consideramos que o estudo de algoritmos de aproximação foi uma boa forma de atingir o objetivo principal desse projeto de iniciação científica,

que era introduzir o aluno à pesquisa na área de otimização combinatória. Para isso, a ideia do projeto foi apresentar ao aluno técnicas de projeto de algoritmos de aproximação, bem como exemplos clássicos de problemas de otimização e algoritmos de aproximação para os mesmos.

A metodologia usada consistiu em estudar o assunto principalmente por meio dos dois principais livros da área, "*Approximation Algorithms*" [6] e "*The Design of Approximation Algorithms*" [7], e realizar reuniões semanais entre o aluno e a orientadora para discutir os tópicos estudados.

Desde o início desse projeto, o aluno já cursou em paralelo as disciplinas Teoria dos Grafos e Análise de Algoritmos, que introduziram muitos conceitos importantes para o estudo de algoritmos de aproximação e para a área de otimização combinatória em geral.

Esse relatório serve como instrumento de avaliação final da pesquisa pela FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), instituição que financia a pesquisa. O cunho do relatório é técnico, procurando sempre documentar os resultados estudados ao longo do período de 01/09/2019 até 31/09/2020.

Outra função importante do relatório foi a consolidação das ideias envolvidas nos algoritmos por parte do aluno, pois foi através do relatório que o aluno organizou as ideias segundo o formalismo necessário para a área, servindo como meio de praticar a escrita científica.

A estruturação escolhida para o relatório, além da presente introdução, baseia-se em capítulo e seções. O Capítulo 1 apresenta diversas definições, notações e conceitos básicos relacionados a área de algoritmos de aproximação e otimização combinatória em geral, sendo que essas áreas estão divididas em seções. Já o Capítulo 2 apresenta algoritmos combinatórios para diversos problemas computacionais, cada seção fala sobre um problema e apresenta ao menos um algoritmo combinatório para o mesmo. O Capítulo 3 define o que são esquemas de aproximação e apresenta dois problemas, um em cada seção, e esquemas de aproximação que podemos usar nesses problemas.

O Capítulo 4 faz uma apresentação sobre a programação linear e diversos conceitos relacionados à mesma, como a dualidade, e também fala sobre a relevância do tópico para o projeto de algoritmos. No Capítulo 5 é feita a apresentação de alguns conceitos de probabilidade e como os mesmos podem ser relacionados ao projeto de algoritmos, além de apresentarmos algoritmos probabilísticos para o problema da Satisfatibilidade Máxima que usam programação linear. O Capítulo 6 fala sobre a aplicação de técnicas de análise e projeto de algoritmos que usam programação linear e aplicamos tais técnicas ao problema da Cobertura por Conjuntos, apresentado no Capítulo 2. O relatório é finalizado com as considerações finais sobre o trabalho, onde falaremos como os objetivos do projeto foram trabalhados ao longo do seu desenvolvimento e qual foi a influência do projeto na formação do aluno.

Capítulo 1

Conceitos Básicos

O objetivo desse capítulo é apresentar conceitos e notações que serão usados ao longo desse relatório. Para isso, dividimos o capítulo em quatro seções: Teoria dos Grafos (1.1), Complexidade Computacional (1.2), Problemas de Otimização (1.3) e Algoritmos de Aproximação (1.4).

1.1 Teoria dos Grafos

Nessa seção apresentaremos algumas definições e resultados da teoria dos grafos, muito usados para modelar problemas de otimização. As principais referências usadas nessa seção foram o clássico livro sobre teoria dos grafos de Bondy e Murty [1] e o livro de Introdução a Algoritmos de Aproximação de Carvalho et al. [2].

Definição 1.1 (Grafo)**.** *Um* **grafo simples** *é um par de conjuntos* (V, E)*, onde* V *é um conjunto finito de elementos e* E *é um conjunto de pares não ordenados dos elementos de* V*.*

A definição de grafo ainda pode ser generalizada para o conceito de multigrafo, porém, nesse relatório trabalharemos apenas com grafos simples, de

forma que a Definição 1.1 é suficiente. Assim, quando nos referirmos a um grafo nesse relatório sempre estaremos falando de um grafo simples.

Dado um grafo $G = (V, E)$, chamamos os elementos de V de ***vértices*** do grafo e chamamos os elementos de E de ***arestas*** do grafo. Assim, V é o ***conjunto dos vértices*** de G e E é o ***conjunto das arestas*** de G.

Ao longo desse relatório, para um grafo $J = (A, B)$ qualquer vamos denotar por $V(J)$ seu conjunto de vértices ($V(J) = A$) e $E(J)$ seu conjunto de arestas ($E(J) = B$). Dessa forma, podemos definir um grafo sem precisar explicitar os elementos do par.

Dado um grafo G, denotaremos a aresta formada pelo par $u, v \in V(G)$ por uv.

Definição 1.2 (Extremos de Arestas, Vértices Adjacentes e Incidência de Aresta)**.** *Dado um grafo G e dados $v, w \in V(G)$, sendo que $vw \in E(G)$, dizemos que:*

- *v e w são* **extremos** *de vw;*
- *v e w são vértices* **adjacentes** *ou* **vizinhos***;*
- *A aresta uw* **incide** *em v e em w.*

Definição 1.3 (Grafo Completo)**.** *Dado um grafo G, se $\forall\ v \in V(G)$ temos que v é adjacente a todos os elementos de $V(G) \setminus \{v\}$, então G é um* **grafo completo**.

Para qualquer grafo completo a seguinte proposição é válida.

Proposição 1.1. *Dado um grafo completo G e $n = |V(G)|$, então*

$$|E(G)| = \frac{n(n-1)}{2}\,.$$

Um grafo G completo com $n = |V(G)|$ é denotado por K_n.

Definição 1.4 (Ordem e Tamanho)**.** *Dado um grafo G, o valor $|V(G)|$ é a* **ordem** *de G, enquanto o valor $|E(G)|$ é o* **tamanho** *de G.*

Definição 1.5 (Corte)**.** *Dados um grafo G e um conjunto $X \subseteq V(G)$, dizemos que um conjunto $E' \subseteq E(G)$ é um* **corte** *se toda aresta $uv \in E(G)$ com $u \in X$ e $v \in V(G) \setminus X$ está contida em E'. Também denotamos $E' = \partial_G(X)$.*

Da Definição 1.5 podemos entender que todo corte em G está associado a uma partição dos vértices de G. Por isso, dado $X \subseteq V(G)$, também podemos escrever $\partial_G(X) = \{X, V(G) \setminus X\}_G$ quando conveniente.

Definição 1.6 (Grau do Vértice)**.** *Dados um grafo G e $v \in V(G)$, o* **grau** *de v é o valor da cardinalidade do corte de $\{v\}$ em G. O grau de um vértice v em G é denotado por $deg_G(v)$ ou $d_G(v)$ (ou apenas $deg(v)$ e $d(v)$, quando G for claro do contexto).*

Seguem dois resultados que podem ser considerados fundamentais na área de teoria dos grafos.

Teorema 1.1 (Teorema do aperto de mãos)**.** *Dado um grafo G, o valor da soma dos graus de seus vértices é dado por*

$$\sum_{v \in V(G)} deg(v) = 2|E(G)|\,.$$

Corolário 1.1. *Dado um grafo G qualquer, a quantidade de vértices em G com grau ímpar é sempre par.*

Dado um grafo G, denotamos o valor do ***grau máximo*** por $\Delta(G)$ e denotamos o valor do ***grau mínimo*** por $\delta(G)$.

Definição 1.7 (Subgrafo)**.** *Dados os grafos G e H, dizemos que H é* **subgrafo** *de G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, de forma que $E(H)$ só contenha arestas com ambos os extremos em elementos de $V(H)$. Também denotamos essa relação por $H \subseteq G$.*

Definição 1.8 (Subgrafo Próprio, Gerador e Induzido)**.** *Dados os grafos G e H, sendo H subgrafo de G, dizemos que:*

- *Se $H \neq G$, então H é* **subgrafo próprio** *de G. Podemos denotar essa relação por $H \subset G$;*
- *Se $V(H) = V(G)$, então H é* **subgrafo gerador** *de G;*
- *Se $E(H)$ é o conjunto de todas as arestas de $E(G)$ que têm ambos os extremos em $V(H)$, então H é* **subgrafo induzido** *por $V(H)$.*

Dados um grafo G e $S \subseteq V(G)$, denotamos por $G[S]$ o subgrafo induzido pelos elementos de S. De forma equivalente, dado $T \subseteq E(G)$, denotamos por $G[T]$ o subgrafo induzido pelos vértices que são extremos de algum elemento de T.

Definição 1.9 (Passeio)**.** *Dado um grafo G, chamamos de* **passeio** *em G uma sequência $W = (v_0, e_1, v_1, \ldots, e_k, v_k)$, onde $\forall\ 0 \leq i \leq k$ vale que $v_i \in V(G)$, e $\forall\ 0 < j \leq k$ vale que $e_j = v_{j-1}v_j \in E(G)$.*

Definição 1.10 (Passeio aberto e fechado)**.** *Dados um grafo G e um passeio $W = (v_0, e_1, v_1, \ldots, e_k, v_k)$ em G, dizemos que o passeio W é* **aberto** *se $v_0 \neq v_k$ e é* **fechado** *se $v_0 = v_k$.*

Definição 1.11 (Trilha)**.** *Dados um grafo G e um passeio $T = (v_0,\ e_1,\ v_1,\ \ldots,\ e_k,\ v_k)$ em G, se para todo par (i, j), onde $i \neq j$ e $0 < i, j \leq k$, vale que $e_i \neq e_j$, então chamamos T de* **trilha**.

Definição 1.12 (Caminho)**.** *Dados um grafo G e um passeio $P = (v_0\ , e_1,\ v_1,\ \ldots, e_k, v_k)$ em G, se para todo par (i, j), onde $i \neq j$ e $0 \leq i, j \leq k$, vale que $v_i \neq v_j$, então chamamos P de* **caminho**.

Definição 1.13 (Ciclo)**.** *Dados um grafo G e um caminho $P = (v_0,\ e_1,\ v_1,\ \ldots,\ e_k,\ v_k)$ em G, sendo que $e_{k+1} = v_0v_k \in E(G)$, chamamos $C = (v_0,\ e_1,\ v_1,\ \ldots,\ e_k,\ v_k,\ e_{k+1},\ v_0)$, de* **ciclo**.

Para um grafo G, simples, é comum omitirmos as arestas ao representarmos passeios, trilhas, caminhos e ciclos, em G, ou seja, representamos eles apenas por uma sequência de vértices. Isso é possível, pois entre dois vértices quaisquer em G só pode existir no máximo uma aresta.

Veja que trilhas, caminhos e ciclos são passeios. Assim, quando falamos de alguma nomenclatura ou propriedade específica para passeios, elas podem ser estendidas para trilhas, caminhos e ciclos.

Para qualquer passeio $W = (v_0, \ldots, v_k)$ chamamos v_0 de ***início*** do passeio e v_k de ***término*** do passeio.

Por mais que passeios em um grafo G sejam definidos como uma sequência de vértices, é usual tratarmos eles como subgrafos de G. Assim, um passeio $W = (v_0, \ldots, v_k)$ pode ser visto como um grafo W onde $V(W) = \{v_0, \ldots, v_k\}$ e $E(W) = \{v_0v_1, \ldots, v_{k-1}v_k\}$.

Existem certos passeios, trilhas, caminhos e ciclos, que possuem características especiais e por isso possuem uma nomenclatura própria.

Definição 1.14 (Trilha Euleriana Fechada e Aberta)**.** *Dados um grafo G e uma trilha $T = (v_0, v_1, \ldots, v_k)$, se $\forall\ uv \in E(G)$ é verdade que $uv \in E(T)$, então T é uma* **trilha euleriana***, sendo uma trilha euleriana* **fechada** *se $v_0 = v_k$ e uma trilha euleriana* **aberta** *se $v_0 \neq v_k$.*

Dizemos que se um grafo G possui uma trilha euleriana fechada, então G é um ***grafo euleriano***. Segue um teorema importante sobre grafos eulerianos.

Teorema 1.2. *Um grafo conexo G é euleriano se e somente se todo vértice de G possui grau par.*

Definição 1.15 (Passeio Gerador)**.** *Dados um grafo G e um passeio $W = (v_0, v_1, \ldots, v_k)$ em G, se $\forall\ v \in V(G)$ é verdade que $v \in V(W)$, então W é um* **passeio gerador** *de G.*

Em particular, se um caminho é gerador o chamamos de ***caminho hamiltoniano***, e se um ciclo é gerador o chamamos de ***ciclo hamiltoniano***.

Definição 1.16 (Grafo Conexo)**.** *Dado um grafo G, se para qualquer par $u, w \in V(G)$ podemos construir um passeio com início em u e término em w, então G é um grafo* **conexo**.

Dada a definição de um grafo conexo é fácil perceber o seguinte fato.

Fato 1.1. *Um grafo G é conexo se e somente se $\forall\ X \subset V(G)$ é verdade que $\partial_G(X) \neq \emptyset$.*

Um outro conceito muito importante em teoria dos grafos é o de ***maximalidade***.

Definição 1.17 (Maximal)**.** *Dados os grafos G e H, sendo $H \subseteq G$, dizemos que H é* **maximal** *em relação a uma propriedade P se não existe $H' \subseteq G$ tal que $H \subset H'$ e H' possui a propriedade P.*

Definição 1.18 (Componente)**.** *Dados os grafos G e H, sendo que H é subgrafo de G. Se H é maximal em relação à conexidade, então H é uma* **componente (conexa)** *de G.*

Um grafo que não é conexo é composto por várias componentes conexas. Segue um resultado fundamental relacionado à conexidade.

Proposição 1.2. *Se um grafo G é conexo, então $|E(G)| \geq |V(G)| - 1$. Se G não é conexo, então $|E(G)| \geq |V(G)| - c$, onde c é a quantidade de componentes conexas de G.*

Definição 1.19 (Floresta)**.** *Dado um grafo F, se não existe nenhum ciclo em F, então F é uma* **floresta**.

Definição 1.20 (Árvore)**.** *Dada uma floresta T. Se T é um grafo conexo (ou, equivalentemente, contém apenas uma componente conexa), então T é uma* **árvore**.

Note que com a definição acima é possível perceber que toda componente de uma floresta é uma árvore. Seguem dois resultados que envolvem árvores e florestas.

Lema 1.1. *Dada uma árvore T, então $|E(T)| = |V(T)| - 1$.*

Teorema 1.3. *Dada uma floresta F, então $|E(F)| = |V(F)| - c$, onde c é a quantidade de componentes de F.*

Definição 1.21 (Floresta Geradora e Árvore Geradora)**.** *Sejam os grafos G e H, em que H é subgrafo gerador de G. Se*

- *H é floresta, então H é uma* **floresta geradora**.
- *H é árvore, então H é uma* **árvore geradora**.

Segue um fato sobre grafos conexos.

Fato 1.2. *Todo grafo conexo possui ao menos um subgrafo que é uma árvore geradora.*

Definição 1.22 (Bipartição e Grafo Bipartido)**.** *Seja G um grafo. Se podemos particionar $V(G)$ em dois conjuntos X e Y onde $\partial_G(X) = \partial_G(Y) = E(G)$, então dizemos que (X, Y) é* **bipartição** *de G, além de dizermos que G é um* **grafo bipartido**.

Se temos um grafo G bipartido, onde $V(G)$ é particionado em X e Y, denotamos que G é bipartido através de $V(G) = (X, Y)$.

Para diversos problemas que possuem como parte de sua instância um grafo G, é comum existir uma função de custo nas arestas. Esta seria uma função $c\colon E(G) \to X$, onde X é algum conjunto numérico. De forma análoga, também pode haver uma função de custo nos vértices, onde o domínio da função seria $V(G)$.

Dados um conjunto S, um conjunto finito $X \subseteq S$, um conjunto numérico Y e uma função $f : S \to Y$, sendo que $X \subseteq S$, usaremos a seguinte notação, definida naturalmente:

$$f(X) = \sum_{x \in X} f(x)\,.$$

1.2 Complexidade Computacional

Nessa seção falaremos sobre conceitos básicos de complexidade computacional que serão usados no decorrer do relatório. As principais referências usadas nessa seção foram Introdução aos Algoritmos de Cormen et al. [3] e Introdução a Algoritmos de Aproximação de Carvalho et al. [2].

De maneira informal, um problema é uma tarefa ou uma questão que desejamos resolver, sendo que para resolvermos um problema computacionalmente é necessário descrever seus dados. Para realizar essa descrição usamos uma ***palavra***, que nada mais é do que uma sequência de caracteres.

Todo problema quando resolvido devolve uma solução para uma determinada ***instância***, que é um conjunto de dados de entrada. Dessa forma, para representarmos uma instância em um problema computacional usamos palavras. Cada instância de um problema é representada por uma palavra diferente, assim, o ***tamanho de uma instância*** é o tamanho da palavra que a representa.

Dentre os tipos de problema, existem dois que comentaremos aqui de forma simplificada. Um tipo são os ***problemas de decisão***, que consistem em problemas cuja a resposta é sim ou não. O outro tipo são os ***problemas de otimização***, que buscam uma melhor solução dentro de um conjunto de soluções possíveis.

Para resolvermos um problema através de um computador usamos ***algoritmos***. Um algoritmo pode ser entendido como um sequência de passos que seguem regras bem definidas, que recebem uma entrada e produzem uma saída após a sequência de passos.

Veja que podemos entender a entrada de um algoritmo como a instância de um problema, e sua saída pode ser entendida como a uma possível solução para esse problema.

Ao falarmos que um algoritmo está ***correto***, queremos dizer que para qualquer instância possível o algoritmo devolve como saída uma solução para aquela instância.

Podemos usar algoritmos diferentes para resolver um mesmo problema, de forma que é interessante analisar qual algoritmo resolve o problema de maneira mais eficiente. Para quantificarmos a ***eficiência*** de um algoritmo, usamos a quantidade de operações elementares que ele executa. Chamaremos essa medida de operações elementares de ***tempo de execução*** do algoritmo.

Dado um problema qualquer, quando nos referimos à ***complexidade*** desse problema estamos falando de quão, inerentemente, é difícil resolver esse problema. Agora, se falamos da ***complexidade de um algoritmo*** em específico, estamos falando do tempo de execução desse algoritmo.

Perceba que, conforme o tamanho da entrada de um algoritmo cresce, a quantidade de operações elementares, usadas para processar os dados de entrada, também cresce. Assim, podemos afirmar que o tempo de execução do algoritmo é uma função no tamanho da entrada.

Para analisarmos o tempo de execução, usamos a ***notação assintótica***, cuja definição é dada a seguir.

Definição 1.23 (Notação assintótica)**.** *Dadas as funções positivas $f(n)$ e $g(n)$, dizemos que*

- $f(n) = \Omega(g(n))$ *se existem as constantes positivas c e n_0 tais que,* $c \cdot g(n) \leq f(n)$, $\forall\ n \geq n_0$.
- $f(n) = O(g(n))$, *se existem as constantes positivas k e n_0 tais que,* $k \cdot g(n) \geq f(n)$, $\forall\ n \geq n_0$.
- $f(n) = \Theta(g(n))$, *se existem as constantes positivas c, k e n_0 tais que,* $c \cdot g(n) \leq f(n) \leq k \cdot g(n)$, $\forall\ n \geq n_0$.

A notação mais usual para representar tempo de execução é a "grande O". Isso ocorre pois quando analisamos um algoritmo, o tempo de execução no pior caso é suficiente para uma boa previsão do tempo de execução do algoritmo, de forma que procuramos por um limite superior no mesmo. Segue um exemplo de seu uso.

Exemplo 1.1. *Seja n uma variável que represente o tamanho da instância de um determinado problema. Seja $f(n) = n^2 + 2n + 4$ uma função que descreve o tempo de execução do pior caso de um determinado algoritmo que resolve esse problema. Considere a expressão*

$$c \cdot n^2 \geq n^2 + 2n + 4 \Rightarrow c \geq \frac{n^2 + 2n + 4}{n^2} = 1 + \frac{2}{n} + \frac{4}{n^2}\,.$$

Veja que $\forall\ n \geq 2$,

$$c \geq 1 + \frac{2}{2} + \frac{4}{4} = 3$$

ou seja, se $c \geq 3$ e $n \geq 2$, então $c \cdot n^2 \geq f(n)$. Pela Definição 1.23, podemos concluir que $f(n) = O(n^2)$.

Se temos um algoritmo cuja função de consumo de tempo é $f(n)$ no pior caso, podemos assim falar que esse algoritmo possui complexidade tempo da ordem de n^2, ou simplesmente $O(n^2)$.

Munidos da notação assintótica, podemos definir o que é um algoritmo de tempo polinomial.

Definição 1.24 (Algoritmo Polinomial)**.** *Dados um algoritmo A e uma função $f(n)$ que representa o tempo de execução de A sobre qualquer instância de tamanho n, se $f(n) = O(n^k)$, então A é um* **algoritmo polinomial**.

Dizemos que um algoritmo é ***eficiente*** se ele possui tempo de execução polinomial no pior caso, ou seja, todo algoritmo eficiente é polinomial e vice-versa. Além disso, é comum só falarmos que o algoritmo é polinomial, ao invés de falar que ele possui tempo de execução polinomial no pior caso.

Com esses conceitos podemos definir a classe de problemas $\mathcal{P}$.

Definição 1.25 (Classe $\mathcal{P}$)**.** *Dizemos que um problema de decisão está contido na* **classe** $\mathcal{P}$ *de problemas, se ele pode ser resolvido por um algoritmo eficiente.*

Além da classe de problemas $\mathcal{P}$ também temos a classe de problemas $\mathcal{NP}$, mas antes de defini-la precisamos das definições de algoritmo verificador e certificado positivo.

Definição 1.26 (Algoritmo Verificador e Certificado Positivo)**.** *Dados um problema X de decisão e um algoritmo A, dizemos que A é um* **algoritmo verificador** *se*

1. *Para toda instância I de X cuja a resposta é sim, existe um conjunto de dados D tal que $A(I, D)$ devolve sim; e*

2. *Para toda instância I de X cuja a resposta é não, para qualquer conjunto de dados D vale que $A(I, D)$ devolve não.*

Chamamos o conjunto de dados D, que atende as duas condições acima, de **certificado positivo**.

Definido o que é um algoritmo verificador, segue a definição da classe de problemas $\mathcal{NP}$.

Definição 1.27 (Classe $\mathcal{NP}$)**.** *Dizemos que um problema de decisão está contido na* **classe** $\mathcal{NP}$ *de problemas se para esse problema existe um algoritmo verificador que aceita um certificado positivo em tempo polinomial.*

O que a definição da classe $\mathcal{NP}$ diz no fundo é que, dado um problema de decisão, se existe um algoritmo A polinomial e um conjunto de dados auxiliares D que sempre possam "verificar" se a instância do problema é sim ou não, então tal problema está em $\mathcal{NP}$.

É fácil perceber que $\mathcal{P} \subseteq \mathcal{NP}$, pois sempre é possível verificar em tempo polinomial se um problema em $\mathcal{P}$ é uma instância sim ou não, já que é possível encontrar a resposta em si em tempo polinomial.

Pode parecer óbvio que $\mathcal{P}$ seja apenas uma parte de $\mathcal{NP}$, porém, ainda não existe demonstração de que $\mathcal{P} \neq \mathcal{NP}$. Assim, surge um dos maiores problemas da matemática contemporânea.

Problema 1.1. *$\mathcal{P}$ é igual a $\mathcal{NP}$?*

De fato, o Problema 1.1 é tão importante que está na lista dos problemas do milênio[1].

Existem ainda outras classes de problemas associadas à classe $\mathcal{NP}$, mas antes de falarmos sobre elas vamos falar sobre redução entre problemas.

Definição 1.28 (Redução)**.** *Sejam P_1 e P_2 dois problemas de decisão. Se existe uma forma de adaptar uma instância qualquer α de P_1 para uma instância β de P_2, de forma que a resposta de P_2 para instância β é "sim" se e somente se a resposta de P_1 para a instância α é "sim", então chamamos o processo de adaptação de instância em conjunto com a resolução de P_1 para a instância β de* ***Redução****.*

Se existe uma redução do problema P_1 para o problema P_2, então dizemos que P_1 pode ser reduzido para P_2. Da definição de redução segue o seguinte fato.

Fato 1.3. *Dados dois problemas P_1 e P_2, se P_1 se reduz a P_2, então a complexidade de P_1 é no máximo equivalente à complexidade de P_2.*

Dado o conceito de redução entre problemas, seguem as definições das classes $\mathcal{NP}$-difícil e $\mathcal{NP}$-completo.

Definição 1.29 (Classe $\mathcal{NP}$-completo)**.** *Um problema X pertence à classe* **$\mathcal{NP}$-completo** *se*

1. *$X \in \mathcal{NP}$; e*
2. *$\forall\ Y \in \mathcal{NP}$ é verdade que Y se reduz a X.*

Definição 1.30 (Classe $\mathcal{NP}$-difícil)**.** *Um problema X pertence à classe* **$\mathcal{NP}$-difícil** *se $\forall\ Y \in \mathcal{NP}$ é verdade que Y se reduz a X.*

[1] http://www.claymath.org/millennium-problems

Perceba que todo problema contido em $\mathcal{NP}$-completo também está contido em $\mathcal{NP}$-difícil, mas o contrário não é verdade.

Da definição da classe $\mathcal{NP}$-completo segue o teorema abaixo.

Teorema 1.4. *Dado um problema $X \in \mathcal{NP}$-completo, então $\mathcal{P} = \mathcal{NP}$, se e somente se, $X \in \mathcal{P}$.*

1.3 Problemas de Otimização Combinatória

Problemas de otimização visam encontrar uma solução ótima dentro de várias soluções viáveis para uma dada instância. Já os problemas de otimização combinatória são os problemas de otimização cuja as variáveis possuem domínios discretos. Assim, se faz necessário estabelecer algum critério para determinar o quão boa uma solução é.

Com essa breve descrição, podemos assumir que, embora os problemas de otimização sejam muito diversos entre si, todos eles possuem alguns elementos em comum [2].

Conforme discutido na seção anterior, a noção de problema que estamos usando sempre envolve um conjunto de dados de entrada para qual o problema deve ser resolvido. Aqui surge o primeiro elemento comum a todo problema de otimização, uma ***instância de entrada***.

Como o que estamos buscando é uma solução ótima para uma dada instância de entrada, então existem diversas soluções possíveis para essa mesma instância, ou seja, existem soluções que também resolvem o problema, porém não são ótimas. Assim, o segundo elemento dos problemas de otimização é o ***conjunto de soluções viáveis*** para uma determinada instância.

É claro que se estamos buscando uma solução ótima dentro das soluções viáveis, então, existe um ***critério de otimização***, sendo esse o terceiro elemento comum aos problemas de otimização. Podemos enxergar esse critério

de otimização como uma função que atribui um valor para cada solução viável.

Então podemos concluir que todo problema de otimização P apresenta:

1. Um conjunto de dados I_P que define uma instância de entrada;

2. Um conjunto de soluções viáveis $S_P(I_P)$ para cada instância de entrada I_P; e

3. Uma função $val_P(S)$ que indica quão boa é uma solução viável $S \in S_P(I_P)$.

Além dos itens acima, todo problema de otimização apresenta uma indicação se ele é de minimização ou de maximização.

Definição 1.31 (Instância Inviável)**.** *Dado um problema P e uma instância I desse problema, se $S_P(I) = \emptyset$, chamamos I de* ***instância inviável*** *de P.*

Outra coisa importante de considerar-se em um problema de otimização é uma solução viável com valor ótimo, conhecida também como ***solução ótima***.

Como dito acima, dentro dos problemas de otimização temos aqueles que são de maximização e aqueles que são de minimização. Um problema de otimização P é dito ***problema de minimização*** se ele possui como solução ótima a solução viável com menor valor de val_P. Por outro lado, P é dito ***problema de maximização*** se sua solução ótima fosse a solução viável com maior valor de val_P.

Uma característica comum a diversos problemas de otimização é que eles possuem um domínio de soluções viáveis muito grande, mas finito, para uma dada instância. Dessa forma, do ponto de vista computacional, é inviável tentar resolvê-los por algoritmos de força bruta, que têm como estratégia testar todas as soluções viáveis para decidir qual é a ótima, pois não seriam eficientes.

Dados um algoritmo A que resolve um problema P e uma instância I de P, representaremos por $A(I)$ a solução viável de P devolvida por A para a instância I.

Outra característica de problemas de otimização é que muitos são $\mathcal{NP}$-difíceis, de forma que não há esperança em encontrar algoritmos que resolvam de maneira exata tais problemas em tempo polinomial, ou seja, não espera-se encontrar algoritmos eficientes para resolvê-los.

Uma estratégia usada para abordar problemas de otimização que sejam $\mathcal{NP}$-difíceis são os algoritmos de aproximação, tema central desse relatório que é apresentado na próxima seção.

Para finalizar essa seção temos três notações que frequentemente serão usadas ao longo do relatório, que são:

- I para representar a instância do problema que está sendo tratado;
- OPT_P para representar o valor de uma solução ótima do problema P;
- Um conjunto de valores entre $\langle\rangle$ é a instância de algum problema.

Lembrando que as notações acima só devem ser consideradas se na mesma seção nenhum outro significado for atribuído a elas.

1.4 Algoritmos de Aproximação

Como já dito na seção anterior, muitos problemas de otimização são $\mathcal{NP}$-difíceis, de forma que não há esperança em encontrarmos algoritmos eficientes que os resolvam. Uma abordagem usada para contornar essa dificuldade é o uso de ***algoritmos de aproximação***.

A ideia central dos algoritmos de aproximação é sacrificar a capacidade de encontrar a solução ótima em prol da eficiência, ou seja, o algoritmo dá um resultado com valor aproximado ao valor ótimo, mas tem tempo de execução polinomial.

Definição 1.32 (Algoritmo de Aproximação e Fator de Aproximação)**.** *Considere um problema de otimização P, um algoritmo A que encontre uma solução viável para P, uma instância I de P e um valor $\alpha > 0$. Dizemos que A é um* **algoritmo de aproximação** *para P se*

- *P é um problema de minimização, $\alpha \geq 1$ e $val_P(A(I)) \leq \alpha \cdot OPT_P(I)$ para toda instância I; ou se*
- *P é um problema de maximização, $\alpha \leq 1$ e $val_P(A(I)) \geq \alpha \cdot OPT_P(I)$ para toda instância I.*

Em ambos os casos acima chamamos α de **fator de aproximação** *de A.*

Se um algoritmo de aproximação A apresenta fator de aproximação α, então A é uma α-aproximação para o problema que A trata.

Observe que qualquer fator de aproximação pode ser definido por de uma função dependente do tamanho da instância. Em geral, os fatores constantes são mais desejados.

Embora pareça algo consideravelmente mais simples do que a resolução exata do problema, em alguns casos a dificuldade de aproximar o problema é igual à de resolver o problema [2].

Para provar que um algoritmo é de aproximação, basta mostrar a relação apresentada na Definição 1.32. Porém, como não sabemos o valor de uma solução ótima (caso contrário poderíamos resolver o problema), a estratégia padrão é encontrar algum limitante para o valor da solução devolvida pelo algoritmo que também possa ser relacionado ao valor de uma solução ótima do problema. Então, por exemplo, para um problema de minimização, tentamos mostrar que $val_P(A(I) \leq \alpha L$ e que $L \leq \beta OPT_P(I)$ para poder dizer que o algoritmo A é uma $\alpha\beta$-aproximação.

Um tópico de extrema relevância na análise e projeto de algoritmos de aproximação é a identificação de estruturas computacionais que fornecem limitantes para o valor da solução ótima do problema.

Capítulo 2

Algoritmos Combinatórios

Nesse capítulo falaremos sobre alguns problemas de otimização clássicos e apresentaremos algoritmos de aproximação para tais problemas. A organização do capítulo foi feita de forma que cada seção apresente um problema e um ou mais algoritmos de aproximação para o mesmo, geralmente de acordo com o seguinte roteiro de apresentação:

1. Apresentação do problema conceitualmente;
2. Apresentação do problema formalmente;
3. Apresentação do pseudocódigo do algoritmo de aproximação;
4. Explicação do funcionamento do algoritmo;
5. Demonstração do fator de aproximação do algoritmo;
6. Análise da complexidade de tempo do algoritmo;
7. Exemplo de aplicação do algoritmo sobre uma instância do problema.

Algumas seções ainda apresentam demonstrações sobre a inaproximabilidade do problema em questão. Lembrando que a nomenclatura de ***algoritmo combinatório*** deriva do não uso de programação linear em sua estrutura, uma técnica que discutiremos no Capítulo 4.

2.1 Escalonamento de Tarefas (*Makespan Scheduling*)

O primeiro problema para o qual mostraremos um algoritmo de aproximação é o escalonamento de tarefas em máquinas idênticas. Esse é um problema clássico que consiste em dividir entre várias máquinas idênticas tarefas com tempos de execução pré-determinados, de forma que o tempo máximo que qualquer máquina executará as tarefas seja o mínimo possível. O Problema 2.1 formaliza essa descrição.

Problema 2.1 (Escalonamento de Tarefas em Máquinas Idênticas)**.** *Dados* $n, m \in \mathbb{N}$ *e* $t : \{1, \ldots, n\} \to \mathbb{R}^+$*, desejamos encontrar uma partição de* $\{1, \ldots, n\}$ *em* m *conjuntos,* $M_1, \ldots, M_m$*, de forma a minimizar o valor*

$$\max_{j \in \{1,\ldots,m\}} \left\{ \sum_{i \in M_j} t(i) \right\}.$$

Observando o Problema 2.1, é possível notar que se trata de um problema de minimização e que uma instância do problema é composta por três elementos:

- Valor m, que indica a quantidade de máquinas, sendo que $\forall\ j \in \{1, \ldots, m\}$ cada conjunto M_j armazena as tarefas alocadas à máquina j;
- Valor n, que indica a quantidade de tarefas, sendo que $i \in \{1, \ldots, n\}$ é uma tarefa;
- Função t, que associa cada elemento de $i \in \{1, \ldots, n\}$ a um valor $t(i) \in \mathbb{R}^+$, indicando o custo de cada tarefa.

Assim, quando falarmos de uma instância I do problema estaremos falando de uma instância na forma $I = \langle m, n, t \rangle$.

Quando falamos do "custo de uma máquina M", nos referimos ao valor $\sum_{i \in M} t(i)$, que será naturalmente representado por $t(M)$.

Perceba que dada uma instância I para o problema, uma solução viável qualquer para o problema é uma distribuição das tarefas sobre as máquinas. Já o valor de uma solução é maior custo dentre os custos das máquinas.

O problema do escalonamento de tarefas é $\mathcal{NP}$-difícil caso $m \geq 2$ [2]. Isso porque caso $m = 1$, então não haveria o que otimizar no problema, uma vez que todas as tarefas teriam de ser realizadas por apenas uma máquina.

É importante não confundir esse problema com o escalonamento de tarefas compatíveis, que adicionalmente possui horário de início e fim para cada tarefa e, assim, só algumas tarefas podem ser escolhidas [3].

O primeiro algoritmo de aproximação já desenvolvido visava aproximar, justamente, o problema do escalonamento de tarefas [2] e é uma 2-aproximação para o problema. Ele é conhecido como algoritmo de Graham e aqui é apresentado no Algoritmo 1.

O Algoritmo 1 é guloso e se baseia em alocar cada uma das n tarefas à máquina M_k que, no momento da alocação, possua custo mínimo. Ele retorna o custo total da máquina mais custosa ao fim da distribuição de tarefas.

Algoritmo 1: EscalonamentoGraham(m, n, t)

Entrada: Valores m para a quantidade de máquinas, n para a quantidade de tarefas e função t que associa cada tarefa i a um custo $t(i)$

Saída: Distribuição das n tarefas entre as m máquinas

```
para j = 1 até m faça
    M_j = ∅ /* cria-se m vetores representando as m máquinas */
fim
para i = 1 até n faça
    Seja k ∈ {1, ..., m} tal que M_k é a máquina com t(M_k) mínimo
    M_k = M_k ∪ {i}
fim
retorna max_{j∈{1,...,m}} t(M_j)
```

Vamos agora mostrar duas delimitações para $OPT_{ESC}(I)$, através de dois lemas, que mais tarde serão úteis para demonstrar que de fato o Algoritmo 1 é uma 2-aproximação para o problema.

Lema 2.1. *Seja $I = \langle m, n, t\rangle$ uma instância do problema do escalonamento. Então $OPT_{ESC}(I) \geq \max_{i\in\{1,\ldots,n\}}\{t(i)\}$, ou seja, o custo ótimo é no mínimo o custo da tarefa de maior custo.*

Demonstração. Dada uma instância I, seja k uma tarefa tal que $t(k) = \max_{i\in\{1,\ldots,n\}}\{t(i)\}$. Assuma, por contradição, que $OPT_{ESC}(I) < t(k)$. Isso significa que k não foi alocada a nenhuma máquina, o que é um absurdo, pois $OPT_{ESC}(I)$ é o custo de uma solução ótima para I e uma solução ótima é uma solução viável. □

Lema 2.2. *Se $I = \langle m, n, t\rangle$ é uma instância do problema do escalonamento, então $OPT_{ESC}(I) \geq \frac{1}{m}\sum_{i=1}^{n} t(i)$, ou seja, o custo ótimo é maior ou igual que a média do custo total das tarefas por máquina.*

Demonstração. Como $\frac{1}{m}\sum_{i=1}^{n} t(i)$ representa a média dos custos das tarefas para todas as máquinas e $max_{i\in\{1,\ldots,n\}}\{t(i)\}$ representa a tarefa de maior custo, então a desigualdade $\frac{1}{m}\sum_{i=1}^{n} t(i) \leq \max_{i\in\{1,\ldots,n\}}\{t(i)\}$ é válida. Assim, o resultado segue diretamente do Lema 2.1. □

Vamos agora provar que o Algoritmo 1 é uma 2-aproximação.

Teorema 2.1. *O algoritmo* GrahamEscalonamento*(m, n, t) é uma 2-aproximação para o problema do Escalonamento de Tarefas em Máquinas Idênticas.*

Demonstração. Veja que, para toda iteração em i do segundo laço **para** do Algoritmo 1, como M_k é a máquina de custo mínimo, então no momento da seleção de M_k temos

$$t(M_k) \leq \frac{1}{m}\sum_{j=1}^{m} t(M_j)\,.$$

Vamos considerar que $t(M_k) = \mathcal{L}$ para M_k no momento de sua seleção.

Observe que, independente da iteração em que o algoritmo esteja, sendo I a instância do problema, vale que

$$\frac{1}{m}\sum_{j=1}^{m} t(M_j) \leq \frac{1}{m}\sum_{i=1}^{n} t(i)\,.$$

Além disso, sabemos pelo Lema 2.2 que

$$\frac{1}{m}\sum_{i=1}^{n} t(i) \leq OPT_{ESC}(I)\,.$$

Agora, na linha do Algoritmo 1 logo após a seleção de M_k, perceba que para qualquer i que esteja sendo alocada, vale, pelo Lema 2.1 que

$$t(i) \leq max_{p\in\{1,\ldots,n\}}\{t(p)\} \leq OPT_{ESC}(I)\,.$$

Assim, após a alocação de qualquer tarefa, vale que

$$\mathcal{L} + t(i) \leq 2\,OPT_{ESC}(I)\,.$$

Perceba que essa última desigualdade é válida em todas as atribuições de tarefas, inclusive na atribuição da última tarefa. Assim,

$$\max_{j\in\{1,\ldots,m\}}\{t(M_j)\} \leq 2\,OPT(I)\,,$$

de onde segue o resultado. □

Sobre o tempo consumido pelo Algoritmo 1, veja que esse algoritmo se baseia em dois laços **para**, sendo que um deles é $O(m)$ e o outro é $O(n)$. Como sabemos que $n > m$, então temos que a complexidade do algoritmo é $O(n)$.

Segue abaixo um exemplo de instância para o Problema 2.1, cuja resposta

devolvida pelo Algoritmo 1 é assintótica em relação ao fator de aproximação.

Exemplo 2.1. *Seja* $I = \langle m, n, t \rangle$ *uma instância do problema do escalonamento de tarefas, onde* $m = 3$ *é a quantidade de máquinas,* $n = 10$ *é a quantidade de tarefas e* t *é a função de custo para as tarefas, definida da seguinte forma: as tarefas de índices* 1 *a* 9 *possuem custo unitário e a tarefa de índice* 10 *possui custo* 3*.*

Da forma que as tarefas estão ordenadas, após as 9 *primeiras iterações do segundo laço* ***para*** *do Algoritmo 1 o custo de todas as máquinas seria igual a* 3*, sendo que cada máquina conteria três das nove tarefas de custo unitário. Após a alocação da última tarefa uma das três máquinas passaria a ter custo* 6*.*

A solução ótima para essa instância possui custo 4*. A distribuição das tarefas na solução ótima seria* $M_1 = \{7, 10\}$*,* $M_2 = \{1, 3, 5, 8\}$ *e* $M_3 = \{2, 4, 6, 9\}$*. Veja que cada uma das máquinas possui custo* 4*.*

Esse exemplo pode ser generalizado para uma instância do tipo $\langle m, m^2 + 1, t \rangle$ *onde o custo das tarefas de* 1 *a* m^2 *é unitário e o custo da tarefa* $m^2 + 1$ *é* m*. Com isso, temos que o custo da solução devolvida pelo algoritmo é* $2m$ *enquanto que o custo da solução ótima é* $m + 1$*. Assim, assintoticamente o fator de aproximação é* 2*.*

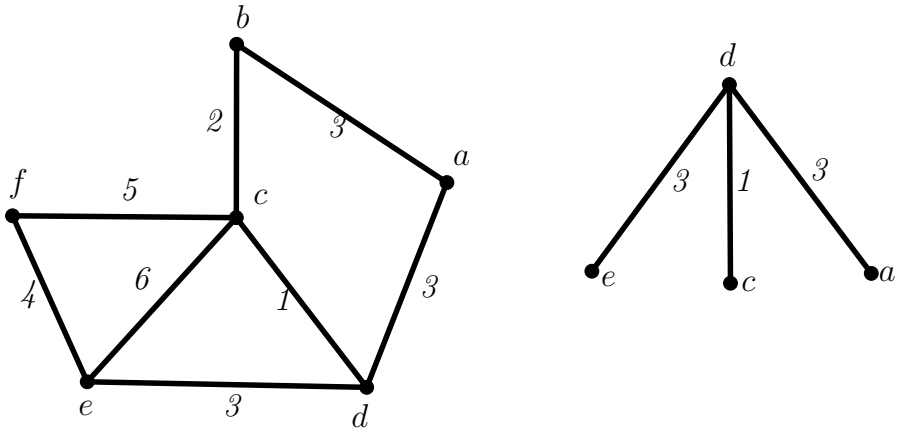
2.2 Árvore de Steiner (*Steiner Tree*)

Imagine que, dado um grafo qualquer com custo nas arestas, queiramos encontrar uma árvore de custo mínimo que conecte um determinado subconjunto dos vértices desse grafo. Basicamente, acabamos de descrever o problema de encontrar uma árvore de Steiner em um grafo. Segue a formalização do problema.

Problema 2.2 (Árvore de Steiner)**.** *Dado um grafo G, uma função $c : E(G) \to \mathbb{R}^+$ e o conjunto $R \in V(G)$, desejamos encontrar uma árvore $T \subseteq G$ tal que $c(T) = \sum_{e \in E(T)} c(e)$ seja mínimo e $R \subseteq V(T)$.*

Chamaremos o conjunto R de conjunto requisitado e o conjunto $S = V(G) \setminus R$ de conjunto Steiner. Segue abaixo um exemplo de um grafo e da árvore de Steiner para um dado conjunto R.

Exemplo 2.2 (Árvore de Steiner)**.** *Abaixo seguem dois grafos. Do lado esquerdo temos um grafo G, com todos seus vértices, arestas e custos das arestas representados. À direita temos uma árvore de Steiner para o grafo G, considerando que $R = \{a, c, e\}$, perceba que para construirmos essa árvore também usamos o vértice $d \notin R$.*



O Problema da Árvore de Steiner, da forma como foi descrito, pertence a $\mathcal{NP}$-Difícil [6].

Perceba que uma instância da Árvore de Steiner é dada por um grafo G, uma função c de custo nas arestas e um conjunto R de vértices do grafo. Assim, vamos considerar que $I = \langle G, c, R \rangle$.

Uma variante do Problema da Árvore de Steiner é o Problema da Árvore de Steiner Métrica. A diferença dessa variante é a imposição de duas restrições para a instância do problema:

1. O grafo tem que ser completo; e

2. A função custo das arestas deve seguir a ***desigualdade triangular***, ou seja, dados três vértices u, v, w quaisquer desse grafo vale que $c(uv) \leq c(uw) + c(wv)$, sendo c a função custo.

De forma geral, quando falamos da versão métrica de um problema, que tenha como parte de sua entrada um grafo, queremos dizer que esse grafo segue as duas restrições acima.

Se um grafo G com função de custo c nas arestas segue as duas restrições acima, falaremos que tal grafo é ***métrico***.

Segue uma formalização da versão métrica do Problema da Árvore de Steiner.

Problema 2.3 (Árvore de Steiner Métrica)**.** *Dado um grafo métrico G, uma função $c : E(G) \to \mathbb{R}^+$ e o conjunto $R \in V(G)$, desejamos encontrar uma árvore $T \subseteq G$ tal que $c(T)$ seja mínimo e $R \subseteq V(T)$.*

Veja que a descrição dos dois problemas é praticamente igual, sendo a única variante o fato do grafo da instância ser métrico. Inclusive, a versão métrica da Árvore de Steiner também pertence a $\mathcal{NP}$-difícil.

A razão de estarmos falando da Árvore de Steiner Métrica é porque toda instância de Árvore de Steiner pode ser transformada, em tempo polinomial,

em uma instância de Árvore de Steiner Métrica. Mais do que isso, qualquer aproximação para a Árvore de Steiner Métrica também garante uma aproximação para a Árvore de Steiner. O teorema a seguir prova isso.

Teorema 2.2. *Qualquer fator de aproximação é preservado numa redução da Árvore de Steiner para a Árvore de Steiner Métrica.*

Demonstração. Vamos começar mostrando que sempre podemos transformar uma instância da Árvore de Steiner para uma instância da Árvore de Steiner Métrica em tempo polinomial.

Dados um grafo G, uma função $c : E(G) \to \mathbb{R}^+$ e um conjunto $R \subseteq V(G)$, seja $I_{AS} = \langle G, c, R \rangle$ uma instância da Árvore de Steiner.

Vamos construir G', um grafo completo tal que $V(G') = V(G)$. Dados dois vértices quaisquer $v, w \in V(G')$, a função de custo $c' : E(G') \to \mathbb{R}^+$ é definida de forma que $c'(vw)$ possui o mesmo valor que o passeio de custo mínimo entre v e w em G.

Perceba que dados quaisquer três vértices $v, w, u \in V(G')$, é verdade que $c'(vw) \leq c'(vu) + c'(uw)$ pois, caso contrário, haveria passeio de custo menor entre v e w. Assim, o grafo G' é completo e c' respeita a desigualdade triangular, logo G' é métrico. Assim, $I_{ASM} = \langle G', c', R \rangle$ é uma instância para a Árvore de Steiner Métrica.

Agora, considere que T' seja uma árvore de custo ótimo para o problema da Árvore de Steiner Métrica sobre a instância I_{APM}, isto é, $c'(T') = OPT_{ASM}(I_{ASM})$.

Veja que T' é uma árvore em G' mas não necessariamente em G. Porém, como toda aresta de T' representa um caminho em G, podemos construir um subgrafo de G de custo no máximo o de T' (sem repetição de arestas). Vamos chamar tal subgrafo de H. Assim, $c'(T') \leq c(H)$.

Se retirarmos todos os ciclos de H, obtemos uma árvore T em G que liga todos os vértices em R, tal que $c(T) \leq c'(T')$. Logo, T é solução para a Árvore de Steiner na instância $\langle G, c, R \rangle$.

Como $OPT_{AS}(I_{AS}) \leq c(T)$, isso nos permite concluir que o custo da solução ótima da Árvore de Steiner para I_{AS} é menor ou igual ao custo da solução ótima da Árvore de Steiner Métrica para I_{ASM}. Resumindo:

$$c(T) \leq c(H) \leq c'(T') \Rightarrow OPT_{AS}(I_{AS}) \leq OPT_{ASM}(I_{ASM})\,. \tag{2.1}$$

Agora, fazendo o caminho inverso, seja T uma árvore ótima para a Árvore de Steiner sobre a instância I_{AS}, isto é, $OPT_{AS}(I_{AS}) = c(T)$. Veja que como G' é um grafo completo com os mesmos vértices de G, podemos replicar T exatamente em G', criando a árvore T' tal que $c(T) = c'(T')$. Observe que T' é solução viável para Árvore de Steiner na instância $\langle G', c', R\rangle$.

Como $OPT_{ASM}(I_{ASM}) \leq c'(T')$, isso nos permite concluir que o custo da solução ótima da Árvore de Steiner para I_{AS} é maior ou igual ao custo da solução ótima da Árvore de Steiner Métrica para I_{ASM}. Resumindo:

$$c(T) = c'(T') \Rightarrow OPT_{ASM}(I_{ASM}) \leq OPT_{AS}(I_{AS})\,. \tag{2.2}$$

Concluímos então que qualquer instância para Árvore de Steiner pode ser adaptada para Árvore de Steiner Métrica, assim como uma solução de Árvore de Steiner Métrica pode ser adaptada para uma solução de Árvore de Steiner.

Se A é uma α-aproximação para Árvore de Steiner métrica, então por definição temos $c'(A(I_{ASM})) \leq \alpha OPT_{ASM}(I_{ASM})$. Usando a Equação 2.2, podemos escrever que $c'(A(I_{ASM})) \leq \alpha OPT_{AS}(I_{AS})$. Como a solução $A(I_{ASM})$ pode ser adaptada para uma solução S de I_{AS} em que $c(S) \leq c'(A(I_{ASM}))$, como descrito acima, é fácil ver que A também é α-aproximação para Árvore de Steiner.

Note que o processo de adaptação da instância e da resposta entre os problemas é feito em tempo polinomial. □

Vamos passar a falar sobre aproximações de fato para o problema. Em especial falaremos sobre uma aproximação para o problema da Árvore de

Steiner Métrica. A estratégia que usaremos consiste em encontrar uma árvore geradora de custo mínimo para o grafo $G[R]$[1]. Mais à frente demonstraremos que o custo de tal árvore sempre está a um determinado fator de uma árvore de Steiner de custo ótimo.

Segue abaixo a descrição formal do problema da Árvore Geradora Mínima, também chamado de MST, por ser sigla para o inglês *Minimum Spanning Tree*.

Problema 2.4 (Árvore Geradora Mínima)**.** *Dados um grafo G e uma função $c : E(G) \to \mathbb{R}$, queremos encontrar uma árvore geradora T de G tal que $c(T) = \sum_{e \in E(T)} c(e)$ é mínimo.*

Diferente do problema da Árvore de Steiner, o problema de encontrar uma MST pertence a $\mathcal{P}$ e existem vários algoritmos eficientes que resolvem esse problema. Aqui vamos considerar um dos mais populares e simples, o algoritmo de Kruskal (Algoritmo 2).

Algoritmo 2: KRUSKAL(G, c)

Entrada: Grafo G e função c de custo nas arestas de G
Saída: Árvore geradora de custo mínimo em G

```
1 F = ∅ /* F manterá as arestas da árvore final */
2 Seja A um vetor com todas as arestas de E(G) e seja C um vetor tal
    que C[i] = c(A[i])
3 Ordene o vetor C em ordem crescente e altere a ordem dos elementos
    de A para que c(A[i]) = C[i]
4 para i = 1 até |E(G)| faça
5 |   se G[F ∪ A[i]] não possui ciclos então
6 |   |   F = F ∪ A[i]
7 |   fim
8 fim
9 retorna G[F]
```

Veja que o que o Algoritmo 2 faz é adicionar arestas de menor custo ao conjunto $F \subseteq E(G)$, de forma que no final obtenhamos uma árvore geradora,

[1]Lembre-se que $G[R]$ é o grafo de G induzido pelos vértices em R.

baseada no grafo induzido por F. Essa ação de pegar arestas de menor custo faz com que o Algoritmo de Kruskal seja um algoritmo guloso.

Em relação ao tempo consumido por KRUSKAL, considere que $|V(G)| = n$ e $|E(G)| = m$:

- Na linha 1 é realizada uma operação constante, de tempo $O(1)$;
- Na linha 2 ocorrem duas instanciações com todos os elementos de $E(G)$, que leva tempo $O(m)$;
- Na linha 3 ocorre a ordenação dos vetores A e C. Na nossa versão vamos considerar o método de ordenação MergeSort, que leva tempo $O(m \lg m)$;
- A linha 4 é um laço **para** que executa m vezes, logo leva tempo $O(m)$. Isso também implica que os consumos individuais das linhas dentro desse laço sejam multiplicados por m;
- A linha 5 é uma estrutura de decisão, porém a comparação realizada possui um custo elevado, já que é necessário verificar se há ciclos em $G[F \cup A[i]]$. Para verificar a existência de ciclos podemos usar a busca em profundidade, cujo tempo é $O(n + |F|)$. Como F no final possuirá apenas $n - 1$ arestas, esse tempo acaba sendo de fato apenas $O(n)$. Além disso, a operação de busca é realizada no laço **para**, de forma que essa linha leva tempo $O(nm)$;
- Por fim, temos a linha 6, que pode ser implementada em tempo $O(1)$ também, mas como está no laço **para** leva tempo $O(m)$.

Com a análise acima podemos concluir que o tempo dominante no Algoritmo 2 é $O(nm)$, logo é um algoritmo eficiente.

Segue o teorema que garante que qualquer MST encontrada para $G[R]$ está a no máximo um fator 2 de distância da resposta ótima da Árvore de Steiner Métrica.

Teorema 2.3. *Dada uma instância $I = \langle G, c, R\rangle$, onde G é um grafo, c é uma função de custo nas arestas de G e R é um subconjunto dos vértices de G, o custo de uma MST em $G[R]$ é no máximo $2\,OPT_{ASM}(I)$.*

Demonstração. Seja T uma árvore de Steiner ótima para a instância $I = \langle G, c, R\rangle$ do problema. Lembre-se que T com certeza contém todos os vértices de R, mas também pode conter alguns vértices de $V(G) \setminus R$.

Seja T' o grafo obtido pela duplicação das arestas de T. Note que todos os vértices de T' possuem grau par, fazendo valer para esse grafo o Teorema 1.2, que nos garante que existe uma trilha euleriana ξ em T'.

Observe que

$$c(\xi) = 2\,OPT_{ASM}(I)\,. \tag{2.3}$$

Lembre-se que G é completo, logo, em G sempre existem arestas que conectam dois vértices quaisquer de R.

Vamos agora construir um ciclo H que contenha todos os vértices de R. Para isso, vamos ir adicionando os vértices de R segundo a ordem que eles aparecem na trilha ξ. Como G é completo, independente da sequência obtida, H representa um ciclo.

Como G respeita a desigualdade triangular, não há como o custo de H ser maior que o de ξ, já que $V(H) \subseteq V(\xi)$ e o custo da aresta que liga diretamente dois vértices é sempre o caminho menos custoso entre tais vértices. Logo,

$$c(H) \leq c(\xi)\,. \tag{2.4}$$

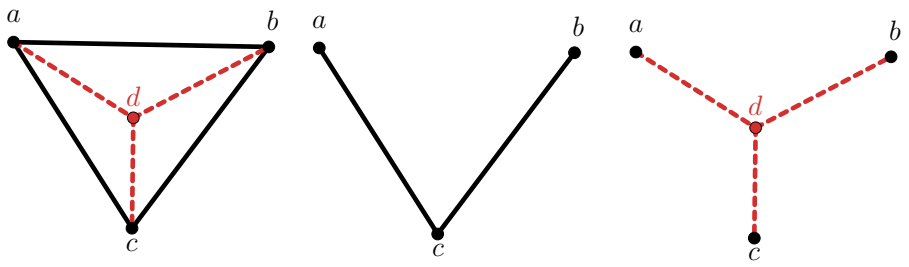
Mas perceba que, além disso, se retirarmos uma aresta de H, obtemos uma árvore geradora A em $G[R]$. Tal árvore certamente tem pelo menos o custo de uma árvore geradora mínima F em $G[R]$, que é a solução devolvida pelo algoritmo. Assim,

$$c(F) \leq c(A) \leq c(H) \leq c(\xi) = 2\,OPT_{ASM}(I)\,. \tag{2.5}$$

□

Provado o fator de aproximação proposto, segue um exemplo que demonstra que ele é justo para o algoritmo.

Exemplo 2.3. *Seguem as representações gráficas de três grafos. O grafo representado mais à esquerda é o grafo G, que é completo. A função c que atribui peso às arestas de G está representada por cores e estilos. As arestas vermelhas tracejadas possuem custo 1 e as arestas pretas sólidas possuem custo 2.*



Dessa forma, se $R = \{a, b, c\}$, então $I_{ASM} = \langle G, c, R\rangle$ é uma instância para Árvore de Steiner Métrica.

Assim, uma resposta ótima para o problema seria a árvore à direita da imagem, com custo total 3. Porém, o algoritmo de aproximação apresentado nessa seção devolveria a árvore no centro da imagem, com custo total 4.

Esse exemplo pode ser generalizado para um grafo com n vértices onde $n-1$ deles estão contidos em R, sendo que o custo das arestas entre os vértices de R é 2 enquanto as arestas com um extremo no vértice de $V(G)\setminus R$ possuem custo 1. A resposta devolvida para esse tipo de instância sempre será $2(n-2)$, sendo que a solução ótima possui custo $n-1$, ou seja, a solução devolvida tem fator assintótica a 2.

2.3 Cobertura por Vértices (*Vertex Cover*)

Uma cobertura por vértices (*vertex cover*) de um grafo é um subconjunto de seus vértices tal que ao menos um extremo de toda aresta pertença a esse subconjunto. A Definição 2.1 formaliza essa ideia.

Definição 2.1 (Cobertura por Vértices)**.** *Seja G um grafo não direcionado. Dizemos que V' é* ***cobertura por vértices*** *de G se $V' \subseteq V(G)$ e $\forall\ uv \in E(G)$ temos que $u \in V'$ ou $v \in V'$.*

Encontrar uma cobertura por vértices qualquer em um grafo não é algo difícil, pois o próprio conjunto de vértices do grafo se encaixa na Definição 2.1. Dito isso, vamos ver o Problema 2.5, que consiste em encontrar uma cobertura por vértices de custo mínimo em um grafo ponderado nos vértices.

Problema 2.5 (Cobertura por Vértices Ponderados)**.** *Dados um grafo não direcionado G e uma função de custo $c\colon V(G) \to \mathbb{Q}^+$, desejamos encontrar uma cobertura por vértices V' para G tal que $\sum_{v \in V'} c(v)$ é mínimo.*

Outro problema associado a encontrar uma cobertura por vértices específica seria encontrar uma cobertura por vértices de cardinalidade mínima, conforme descrito no Problema 2.6.

Problema 2.6 (Cobertura Cardinal por Vértices)**.** *Dado um grafo G, desejamos encontrar uma cobertura por vértices V' para G tal que $|V'|$ é mínimo.*

Veja que o Problema 2.6 pode ser considerado um caso específico do Problema 2.5 em que a função custo possui valor igual para todo vértice. Por isso, para os dois problemas apresentados a instância é dada por

- um grafo simples G;
- uma função custo nos vértices $c : V(G) \to \mathbb{Q}^+$.

Assim, dada uma instância $I = \langle G, c \rangle$, o conjunto de soluções viáveis $S(I)$ envolve todos os subconjuntos de $V(G)$ que podem ser cobertura por vértices do grafo G.

Ambos os problemas de Cobertura por Vértices são $\mathcal{NP}$-difícil.

A partir de agora sempre que nos referirmos ao problema da Cobertura por Vértices estaremos falando do Problema 2.6. Nessa seção mostraremos um algoritmo de aproximação para o mesmo.

Antes de começarmos a dar mais detalhes do algoritmo, vamos nos atentar à Definição 2.2 e à Definição 2.3, que definem o que são um emparelhamento e um emparelhamento maximal.

Definição 2.2 (Emparelhamento)**.** *Dado um grafo G, dizemos que $M \subseteq E(G)$ é um emparelhamento de G se todo par de elementos distintos de M não possui extremos em comum.*

Definição 2.3 (Emparelhamento maximal)**.** *Dado um emparelhamento M de um grafo G, dizemos que M é um emparelhamento maximal se $\forall\ uv \in E(G), M \cup \{uv\}$ não é emparelhamento.*

Dadas essas definições, a estratégia para construção de uma aproximação para o Problema 2.6 é encontrar e devolver o conjunto dos extremos das arestas de um emparelhamento maximal no grafo.

O funcionamento do Algoritmo 3 consiste em:

1. Selecionar uma aresta $uv \in F$ e incluir seus extremos no conjunto C;
2. Excluir de F, que originalmente é igual a E, todas as arestas com extremos em comum com uv;
3. Devolver o conjunto C quando não existirem mais arestas em F.

Veja que o segundo processo descrito acima garante que a Definição 2.2 seja respeitada, enquanto a condição do laço **enquanto** garante que a seleção de vértices respeite a Definição 2.3. Assim, o conjunto C acaba sendo igual

Algoritmo 3: CobVert_Emparelhamento(G)

Entrada: Grafo G
Saída: Cobertura por vértices de G

```
1 C = ∅
2 F = E(G)
3 enquanto F ≠ ∅ faça
4 |   escolha qualquer uv ∈ F
5 |   C = C ∪ {u, v}
6 |   F = F − {arestas com extremo em v ou u}
7 fim
8 retorna C
```

aos vértices extremos de um emparelhamento maximal, o que acaba implicando que C é uma cobertura por vértices, conforme diz o seguinte lema.

Lema 2.3. *Dado um grafo G, o conjunto de vértices de um emparelhamento maximal em G é sempre uma cobertura por vértices de G.*

Demonstração. Seja M um emparelhamento maximal de G, então não existe aresta uv em $E(G)$ tal que $M \cup \{uv\}$ também é emparelhamento. Veja que isso implica que todo elemento em $E(G)$ possui ao menos um extremo em $V(M)$, assim o conjunto $V(M)$ encaixa-se na definição de cobertura por vértices. □

Uma vez definido o Algoritmo 3 usado na resolução do Problema 2.6 vamos demonstrar que ele é uma 2-aproximação. Antes de demonstrarmos o teorema em si vamos voltar nossas atenções a dois lemas. O primeiro lema relaciona os tamanhos de um emparelhamento e uma cobertura por vértices em um mesmo grafo. O segundo lema garante que CobVert_Emparelhamento devolve um emparelhamento maximal.

Lema 2.4. *Dados um grafo G, um emparelhamento M de G e uma cobertura por vértices C de G, sempre é válido que*

$$|C| \geq |M| .$$

Demonstração. Note que qualquer cobertura por vértices, por cobrir todas as arestas de G, deve conter ao menos um vértice extremo de cada aresta do emparelhamento maximal. Se isso não ocorresse, teríamos arestas sem extremos na cobertura por vértices obtida, o que seria contrário à própria definição de uma cobertura por vértices. □

Lema 2.5. *O Algoritmo 3 produz um conjunto de vértices de um emparelhamento maximal.*

Demonstração. Caso existisse alguma aresta que não tivesse um de seus extremos em C, então $F \neq \emptyset$ e tal aresta seria selecionada na linha 4 do algoritmo e seus vértices seriam adicionados em C, que agora configuraria um emparelhamento de G que contém C. □

Teorema 2.4. *O Algoritmo 3 é uma 2-aproximação para o Problema da Cobertura por Vértices.*

Demonstração. Dado o conjunto de vértices C que o Algoritmo 3 retorna para uma instância $G = (V, E)$, considere que M_C é o conjunto das arestas selecionadas na linha 4 ao longo da execução do algoritmo. Pelo Lema 2.5 sabemos que M_C é um emparelhamento maximal. Entao, pelo Lema 2.4, sabemos que

$$OPT_{VC}(G) \geq |M_C| \,.$$

Perceba que C possui todos os vértices das arestas em M_C, ou seja, para cada elemento em M_C temos dois elementos em C. Assim, vale a relação

$$|C| = 2|M_C| \,. \tag{2.6}$$

Dada a Equação (2.6) e o Lema 2.4, concluímos que

$$OPT_{VC}(G) \geq |M_C| = \frac{|C|}{2} \,,$$

de onde temos que

$$2\,OPT(G)_{VC} \geq |C|\,. \tag{2.7}$$

Logo, o Algoritmo 3 é uma 2-aproximação para o problema da Cobertura por Vértices. □

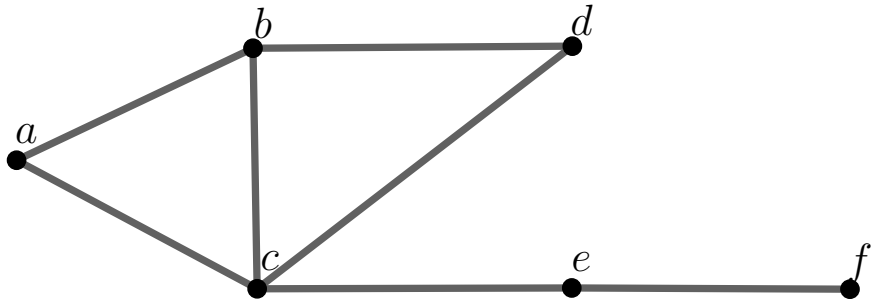
Sendo $n = |V(G)|$ e $m = |E(G)|$, segue uma análise do consumo de tempo do algoritmo linha por linha:

- As linhas 1 e 8 levam tempo $O(1)$;
- A linha 2, considerando uma representação do grafo em matriz de adjacências, tem tempo $O(n + m)$;
- A quantidade de iterações do laço **enquanto** é $O(m)$. Assim, as linhas 4 e 5, que levam tempo $O(1)$, e a linha 6 que leva tempo $O(m)$ a cada iteração, levam no total tempo $O(m)$ e $O(m^2)$.

Dessa forma podemos dizer que o respectivo consumo de tempo do algoritmo é $O(m^2)$ se o grafo for simples e conexo e $O(m + n)$ caso contrário.

Agora que já vimos que de fato o Algoritmo 3 é uma 2-aproximação do Vertex Cover, vamos ver um exemplo de sua aplicação em um grafo.

Exemplo 2.4. *Dado o grafo G em que $V(G) = \{a, b, c, d, e, f\}$ e $E(G) = \{ab, ac, bc, bd, ce, ef\}$, vamos buscar um Vertex Cover para G usando o Algoritmo 3. Esse grafo está graficamente representado a seguir.*



Ao aplicarmos o Algoritmo 3 em G vamos supor que a seguinte ordem de arestas seja tomada na linha 4 a cada iteração:

1. *Seleciona aresta $ab \rightarrow C = \{a, b\}$ e $F = \{cd, ce, ef\}$;*
2. *Seleciona aresta $cd \rightarrow C = \{a, b, c, d\}$ e $F = \{ef\}$;*
3. *Seleciona aresta $ef \rightarrow C = \{a, b, c, d, e, f\}$ e $F = \emptyset$.*

Perceba que $|C| = 6$, enquanto que $OPT_{VC}(G) = 3$, pois basta selecionar os vértices $\{b, c, f\}$ como cobertura de G. Ou seja, obtemos um resultado que respeita a Equação (2.7). Assim, esse exemplo mostra que o fator de aproximação obtido é justo.

2.4 Caixeiro Viajante (*Traveling Salesman*)

Antes de falarmos propriamente do problema do caixeiro viajante vamos formalizar o seguinte problema relacionado.

Problema 2.7 (Ciclo Hamiltoniano)**.** *Dado um grafo G, encontrar um ciclo hamiltoniano em G, isto é, um ciclo gerador em G.*

Normalmente chamado de TSP (acrônimo para *Traveling Salesman Problem*), o problema do Caixeiro Viajante consiste em encontrar um ciclo de custo mínimo, dentro de um grafo ponderado nas arestas que passe em cada um dos vértices desse grafo. Segue abaixo uma formalização do problema.

Problema 2.8 (Caixeiro Viajante)**.** *Dado um grafo G e uma função de custo nas arestas $c : E(G) \to \mathbb{Q}^+$, encontre um ciclo hamiltoniano C em G que minimize o valor de $c(C) = \sum_{e \in E(C)} c(e)$.*

Vamos nos referir ao problema do caixeiro viajante como TSP.

Veja que uma instância do Problema 2.8 é composta por um grafo e função de custo nas arestas de tal grafo.

O problema TSP é $\mathcal{NP}$-difícil [6], então, a não ser que $\mathcal{P} = \mathcal{NP}$, não existe algoritmo que o resolva em tempo polinomial para qualquer grafo. Além disso, o TSP também é inaproximável em tempo polinomial se $\mathcal{P} \neq \mathcal{NP}$, conforme o Teorema 2.5.

Teorema 2.5 (Inaproximabilidade do TSP)**.** *Para toda função $f(n)$ computável em tempo polinomial, o TSP não pode ser aproximado por um fator de $f(n)$, a não ser que $\mathcal{P} = \mathcal{NP}$.*

Demonstração. Vamos realizar uma prova por contradição. Para tal, vamos assumir que existe um algoritmo A para o TSP com fator de aproximação $f(n)$.

A ideia da demonstração é mostrar como resolver o problema do Ciclo Hamiltoniano (Problema 2.7), que sabidamente pertence a $\mathcal{NP}$-completo,

usando o algoritmo A para o TSP, realizando assim uma redução entre os problemas. Ao final, nossa conclusão será de que o TSP possui aproximação polinomial somente se $\mathcal{P} = \mathcal{NP}$.

Vamos começar transformando uma instância do Ciclo Hamiltoniano para uma instância do TSP. Para isso, tome o grafo G com n vértices, instância do Ciclo Hamiltoniano. Construa, a partir de G, um grafo completo G' onde $V(G') = V(G)$ e crie uma função de custos c' nas arestas de G' da seguinte forma:

$$c'(e) = \begin{cases} 1 & \text{se } e \in E(G) \\ f(n) \cdot n & \text{se } e \notin E(G) \end{cases}$$

Assim, adaptamos uma instância $\langle G \rangle$ de Ciclo Hamiltoniano para uma instância $\langle G', c' \rangle$ de TSP, sendo que com certeza há um ciclo hamiltoniano em G', por se tratar de um grafo completo.

Tenha em mente que para construir um ciclo hamiltoniano em um grafo de n vértices são necessárias n arestas. Além disso, por definição, o algoritmo A sempre devolve um ciclo cuja soma dos custos das arestas é pelo menos $OPT_{TSP}(G', c')$ e no máximo $f(n) \cdot OPT_{TSP}(G', c')$.

Perceba que da forma como a função custo c' foi definida, podemos dividir a resposta de A sobre $\langle G', c' \rangle$ em dois casos:

1. Se A devolve um ciclo hamiltoniano C de custo menor que $f(n) \cdot n$, então no grafo G existe ciclo hamiltoniano. Isso porque, com essa resposta, temos certeza que $OPT_{TSP}(G', c') = n$ e, como não é possível que o custo de uma resposta de A seja menor que n, então o ciclo C existe em G.

2. Se A devolve um ciclo hamiltoniano C de custo maior que $f(n) \cdot n$, então no grafo G não existe ciclo hamiltoniano. Isso porque na construção de C foi usada ao menos uma aresta não pertencente a $E(G)$, com custo $f(n) \cdot n$. Dessa forma, o custo de C ultrapassa $f(n) \cdot n$, que seria o valor máximo de uma resposta dada por A caso houvesse um ciclo

hamiltoniano que pudesse ser construído apenas com arestas em $E(G)$.

Com os casos dados acima vemos que o problema do Ciclo Hamiltoniano pode ser respondido para um grafo G em tempo polinomial usando o algoritmo A, que é polinomial, provando que $\mathcal{P} = \mathcal{NP}$.

Concluindo, o problema TSP só é aproximável se $\mathcal{P} = \mathcal{NP}$. □

Provada a inaproximabilidade em tempo polinomial do TSP da maneira como ele está descrito no Problema 2.8, ainda podemos desenvolver um algoritmo de aproximação para o problema, mas temos que impor algumas restrições à instância do problema.

No Problema 2.9, conhecido como TSP Métrico, o objetivo é o mesmo que o do TSP, porém a instância possui certas restrições.

Problema 2.9 (Caixeiro Viajante Métrico)**.** *Dado um grafo G completo e uma função de custo nas arestas $c : E(G) \to \mathbb{Q}^+$ que satisfaz a desigualdade triangular*[2]*, queremos obter um ciclo hamiltoniano C em G, de forma que C possua custo mínimo.*

Vamos nos referir ao problema do Caixeiro Viajante Métrico como TSPM.

Tratando-se de instâncias de entrada, a diferença de uma instância do TSP para o TSPM é que o grafo de entrada é completo e a função de custo nas arestas respeita a desigualdade triangular.

Apresentaremos dois algoritmos de aproximação para o TSPM, sendo o primeiro deles uma 2-aproximação e o segundo uma $\frac{3}{2}$-aproximação. Ambos os algoritmos usam de uma mesma estratégia, variando pontualmente em uma das etapas.

Dado um grafo G completo e uma função c nas arestas de G que respeita a desigualdade triangular, a estratégia para construir uma solução para o TSPM em G usa os seguintes passos:

[2]Conceito apresentado para o Problema 2.3.

1. Construa uma árvore geradora de custo mínimo T do grafo G. Fazemos isso pois $c(T)$ é uma boa delimitação inferior para $OPT_{TSPM}(G, c)$, já que qualquer subgrafo conexo gerador de G (o que inclui a solução ótima do TSPM) vai possuir custo maior ou igual a T. Assim,

$$c(T) \leq OPT_{TSPM}(G, c)\,. \tag{2.8}$$

2. Acrescentamos novas arestas a T, até que todos os vértices possuam grau par, obtendo o grafo T'.

3. Definimos uma trilha euleriana ξ sobre T'.

4. Obtemos um ciclo hamiltoniano de G por meio de *atalhos* em ξ.

A principal diferença entre as duas aproximações que aqui serão apresentadas está no passo 2, ou seja, elas variam na forma de obter T' através da adição de arestas a T até que todos os vértices possuam grau par.

Antes de apresentarmos os algoritmos vamos falar sobre o conceito de atalho que é usado em ambos (passo 4) para obter um ciclo hamiltoniano a partir de uma trilha euleriana.

Seja G um grafo e P um passeio em G. Como P é um passeio, ele pode apresentar repetição de vértices. Um atalho é uma forma de eliminar os vértices repetidos de P "cortando caminho" e construindo uma sequência C de vértices livre de repetições.

De um ponto de vista mais formal sobre atalhos, seja $P = (v_1, \ldots, v_i, a, v_{i+1}, \ldots, v_t)$, onde a representa um vértice que já apareceu em P, isso é, $a = v_j$ para algum $1 \leq j \leq i$. Um atalho em P remove a do passeio, gerando outro passeio $P' = (v_1, \ldots, v_i, v_{i+1}, \ldots, v_t)$, o que é possível se a aresta $v_i v_{i+1}$ existir (e isso é sempre verdade em grafos completos).

Se repetidamente fazemos atalhos sobre uma trilha euleriana em G, podemos construir uma sequência C de vértices que é livre de repetições. Como

G é um grafo completo e a trilha era euleriana, C acaba sendo um ciclo hamiltoniano em G.

O Algoritmo 4 formaliza a forma de gerar um ciclo hamiltoniano C por meio de atalhos em uma trilha euleriana ξ de G.

Algoritmo 4: ATALHO(G, ξ)

Entrada: Trilha euleriana $\xi = (v_1, \ldots, v_m, v_1)$ do grafo G com $m = |E(G)|$ e $n = |V(G)|$
Saída: Ciclo hamiltoniano $C = (w_1, \ldots, w_n, w_1)$

```
1  j = 1
2  i = 1
3  X = ∅
4  enquanto j ≤ n faça
5  |  se v_i ∉ X então
6  |  |  w_j = v_i
7  |  |  X = X ∪ {w_j}
8  |  |  j = j + 1
9  |  fim
10 |  i = i + 1
11 fim
12 retorna C = (w_1, ..., w_n, w_1)
```

Entendido o conceito de atalho e apresentado o Algoritmo 4, vamos agora falar sobre a 2-aproximação para o TSPM.

A forma que a 2-aproximação do TSPM garante que o grau de cada vértice de T' seja par é duplicando todas as arestas presentes na árvore T. O algoritmo em si é chamado de RSL em homenagem a seus autores Rosenkrantz, Stearns e Lewis [2]. Aqui o apresentaremos no Algoritmo 5.

Sejam $|V(G)| = n$ e $|E(G)| = m$. A função MST na linha 1 encontra uma árvore geradora mínima de G, podendo ser, por exemplo, o algoritmo de KRUSKAL. Logo, o consumo de tempo nessa linha pode ser considerado $O(mn)$, conforme visto no Algoritmo 2, na Seção 2.2.

Na linha 2 definimos T' usando os vértices de T e a duplicação das arestas

Algoritmo 5: TSPM-RSL(G, c)

Entrada: Grafo G completo e função c de custo nas arestas de G, que respeita a desigualdade triangular

Saída: Ciclo hamiltoniano em G

1 $T = \text{MST}(G, c)$ /* Encontra árvore geradora mínima de G */
2 $T' = (V(T), E(T) \cup E(T))$ /* T' duplica as arestas de T */
3 $\xi = \textsc{Euler}(T')$ /* Encontra trilha euleriana em T' */
4 $C = \textsc{Atalho}(\xi)$ /* Algoritmo 4 */
5 **retorna** C

também de T. Assim, essa linha consome tempo $O(n + m)$.

Euler, na linha 3, é um algoritmo que encontra uma trilha euleriana em T'. Um exemplo de algoritmo que poderia ser usado é o algoritmo de Fleury, que consome tempo $O(m^2)$.

A função Atalho na linha 4, como já vimos, devolve um ciclo hamiltoniano em G por meio de atalhos em ξ. Vemos que Atalho é um algoritmo eficiente, pois consiste basicamente em um laço que é executado até m vezes, fazendo uma comparação com um conjunto com no máximo n elementos. Logo, Atalho consome tempo $O(mn)$.

Após essa análise linha por linha é possível perceber que o consumo de tempo do Algoritmo 5 é polinomial.

Voltando a falar sobre o algoritmo Atalho, devido à desigualdade triangular de G, o menor custo possível para irmos de um vértice a outro é a aresta que conecta eles diretamente. Logo, Atalho gera um ciclo hamiltoniano com custo certamente menor que o custo de qualquer trilha que o gerou, como a trilha ξ. Isso nos faz chegar ao seguinte lema.

Lema 2.6. *Dados um grafo euleriano G, uma função de custo c nas arestas de G, uma trilha euleriana ξ em G e um ciclo hamiltoniano $C = \textsc{Atalho}(\xi)$, vale que*

$$c(C) \leq c(\xi)\,. \tag{2.9}$$

Agora podemos enunciar o seguinte teorema.

Teorema 2.6 (2-aproximação para o TSPM)**.** *O algoritmo* TSPM-RSL *é uma 2-aproximação para TSPM.*

Demonstração. Como ξ é trilha euleriana derivada da duplicação das arestas da árvore geradora de custo mínimo T, então

$$c(\xi) = 2c(T)\,. \tag{2.10}$$

Além disso, pela estratégia do algoritmo Atalho, o ciclo hamiltoniano C obtido sempre apresenta um custo total menor que o custo total de ξ, conforme a Equação (2.9) nos diz.

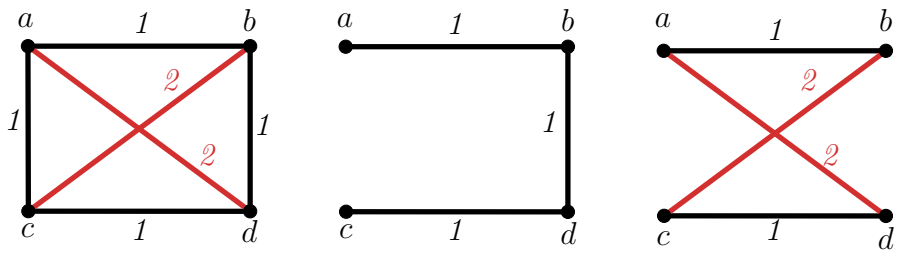
Agora unindo os resultados das expressões (2.8), (2.9) e (2.10) obtemos

$$c(C) \leq c(\xi) = 2c(T) \leq 2OPT_{TSPM}(G, c)\,,$$

de onde vemos que o algoritmo é uma 2-aproximação para TSPM. □

Demonstrada a 2-aproximação do problema TSPM vamos mostrar um exemplo que demonstre que o fator 2 é justo para esse algoritmo.

Exemplo 2.5. *Na figura abaixo, o primeiro grafo é o grafo G. Perceba que ele é um grafo métrico de 4 vértices. Além disso, na própria imagem já estão contidos os custos de cada aresta. Vamos usar G como instância exemplo para o Algoritmo 5.*



O grafo no meio da imagem é a MST T obtida na linha 1 do algoritmo. Suponha que a trilha euleriana obtida após a duplicação de T seja

(b, a, b, d, c, d, b). *Após passar pelo* Atalho *obteríamos como resultado o grafo da direita, que possui custo* 6.

Perceba que para essa instância a solução ótima tem custo $OPT = 4$ *(o ciclo* (a, b, d, c, a)*), porém, o custo da solução obtida é* 6*, que nesse caso é* $2 \cdot OPT - 2$.

Veja que o custo da solução deriva especialmente da trilha euleriana encontrada, então basicamente a seleção de uma trilha euleriana "ruim" leva a uma resposta com custo elevado.

Um tipo de grafo que provaria que o fator de aproximação é justo seria um grafo com $n \geq 6$ *vértices e as seguintes características:*

- *Possuir um ciclo hamiltoniano com todas as arestas com custo* 1*;*
- *Possuir uma estrela com* $n - 1$ *pontas e com todas as arestas possuindo custo* 1*;*
- *O custo de todas as outras arestas é* 2.

Se a MST escolhida pelo algoritmo for a estrela, então o algoritmo sempre pode devolver uma resposta com custo $2 \cdot OPT - 2$ *[6], valor que é assintótico a* $2 \cdot OPT$*, demonstrando que o fator de aproximação é justo.*

Um outro algoritmo de aproximação para o problema do TSPM é o Algoritmo de Christofides. Esse algoritmo possui fator de aproximação $\frac{3}{2}$.

Os passos que o algoritmo de Christofides segue são os mesmos que os do algoritmo RSL, porém, para garantir que o grau dos vértices de T' seja par, usamos um emparelhamento perfeito de custo mínimo em T, apenas sobre os vértices de grau ímpar. Lembrando que a Definição 2.2 explica o que é um emparelhamento, segue a definição de emparelhamento perfeito.

Definição 2.4 (Emparelhamento Perfeito)**.** *Dado grafo* G *dizemos que um emparelhamento* $M \subseteq E(G)$ *é um* **emparelhamento perfeito***, se todo vértice em* G *é extremo de um elemento de* M.

Para encontrar um emparelhamento perfeito de custo mínimo num grafo, já existem algoritmos eficientes, em especial consideraremos o algoritmo de Edmonds. Dado um grafo G, onde $|V(G)| = n$, o Algoritmo de Edmonds encontra um emparelhamento perfeito para G em tempo $O(n^3)$ [2].

O algoritmo de Christofides é apresentado no Algoritmo 6.

Algoritmo 6: TSPM-CHRISTOFIDES(G, c)

Entrada: Grafo G completo e função c de custo nas arestas de G, que satisfaz a desigualdade triangular
Saída: Ciclo hamiltoniano em G

1 $T = \text{MST}(G, c)$ /* Encontra árvore geradora mínima de G */
2 seja I o conjunto de vértices de T com grau ímpar
3 $M = \text{EDMONDS}(G[I], c)$
4 $T' = (V(T), E(T) \cup M)$
5 $\xi = \text{EULER}(T')$ /* Encontra trilha euleriana em T' */
6 $C = \text{ATALHO}(\xi)$ /* Algoritmo 4 */
7 **retorna** C

Veja que o Algoritmo 6 é quase idêntico a TSPM-RSL, sendo a única diferença notável de fato a construção de T', que envolve justamente essa mudança de estratégia para obtermos a garantia de grau par para os vértices.

As linhas 1, 4, 5 e 6 executam funções idênticas àquelas executadas no TSPM-RSL, de forma que o tempo consumido por elas também é igual. A linha 3, como já dito, é executada em tempo $O(n^3)$. Concluindo, o tempo de execução do Algoritmo 6 é polinomial. Segue o teorema que afirma que o Algoritmo 6 é uma $\frac{3}{2}$-aproximação para o TSPM.

Teorema 2.7. *O algoritmo* TSPM-CHRISTOFIDES *é uma* $\frac{3}{2}$*-aproximação para o Problema do Caixeiro Viajante Métrico.*

Demonstração. Como já sabemos pelo Lema 2.6, vale que $c(C) \leq c(\xi)$.

Veja que da forma que T' e, consequentemente, ξ são construídos vale que

$$c(\xi) = c(T') = c(T) + c(M)\,.$$

Agora lembre-se que T é uma árvore geradora de custo mínimo e C é um ciclo hamiltoniano, ambos no mesmo grafo G. Como C sem uma aresta é uma árvore geradora, o custo de C só pode ser maior do que o custo de T. Assim,

$$c(C) \leq c(\xi) = c(T) + c(M) \leq OPT_{TSPM}(G, c) + c(M)\,. \tag{2.11}$$

Vamos construir um ciclo D que seja subsequência da solução ótima. Para construir D, vamos considerar a sequência da solução ótima e selecionar apenas aqueles vértices que possuem grau ímpar em T, ou seja, que estão no conjunto I. Perceba que D é um ciclo hamiltoniano em $G[I]$, pois $G[I]$ é completo.

Pelo Corolário 1.1, sabemos que T possui um número par de vértices com grau ímpar. Assim, D possui um número par de arestas e de vértices. Seja $D = (v_1, v_2, \ldots, v_k)$.

Veja que podemos construir dois emparelhamentos perfeitos M_1 e M_2 para o grafo $G[I]$ da seguinte forma:

- $M_1 = \{v_1v_2, v_3v_4, \ldots, v_{k-1}v_k\}$;
- $M_2 = \{v_2v_3, v_4v_5, \ldots, v_kv_1\}$.

Observe que $D = M_1 \cup M_2$. Como $V(D) = V(G[I])$, temos que

$$2 \cdot c(M) \leq c(M_1) + c(M_2) = c(D)\,, \tag{2.12}$$

afinal, M é emparelhamento perfeito de custo de mínimo para $G[I]$, de forma que os custos de M_1 e M_2 são, com certeza, maiores ou iguais ao de M.

Devido à desigualdade triangular, e por D ter sido construído a partir de uma solução ótima, temos que

$$c(D) \leq OPT_{TSPM}(G, c)\,. \tag{2.13}$$

Com as desigualdades (2.12) e (2.13), podemos concluir que

$$2 \cdot c(M) \leq OPT_{TSPM}(G, c) \Rightarrow c(M) \leq \frac{OPT_{TSPM}(G, c)}{2}\,. \tag{2.14}$$

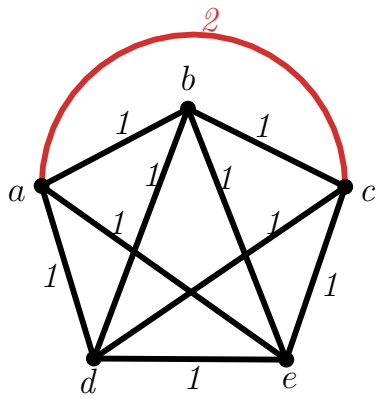
Para finalizar, unindo as desigualdades (2.11) e (2.14) chegamos a seguinte relação:

$$c(C) \leq \frac{3}{2} OPT_{TSPM}(G, c)\,.$$

□

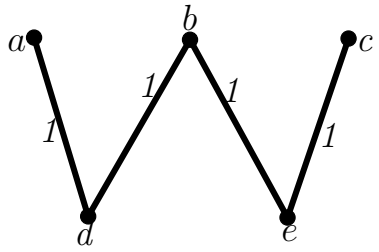
Demonstrado o fator de aproximação do Algoritmo 6, vamos apresentar um exemplo de sua execução.

Exemplo 2.6. *Segue o grafo que usaremos como exemplo, bem como o custo de suas arestas.*

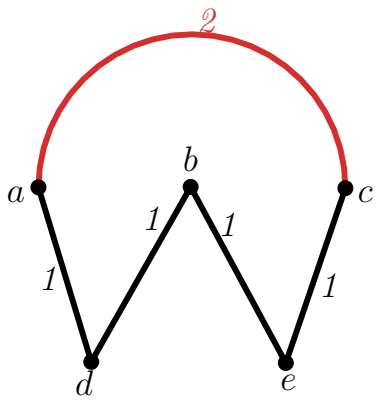


Perceba que todas as arestas do grafo possuem custo 1*, exceto pela aresta que liga os vértices a e c, que possui custo* 2*.*

Como o grafo é métrico, ele pode ser instância dDevido à desigualdade triangular, e por D ter sido construído a partir de uma solução ótima, temos queo algoritmo de Christofides. Suponha que a árvore de custo mínimo, obtida na linha 1 do algoritmo, seja a seguinte.



Dessa forma só temos dois vértices com grau ímpar. Um emparelhamento perfeito de custo mínimo que será escolhido na linha 3 é a aresta que conecta esses dois vértices. Adicionando tal aresta à T para gerar uma trilha euleriana sobre o novo grafo, temos que o ciclo hamiltoniano devolvido ao final do algoritmo será como o abaixo.



Perceba que o custo dessa solução é igual a $(n-1)+\lfloor n/2 \rfloor = 6$, onde n é a quantidade de vértices do grafo. Veja que, nesse caso, $OPT = 5$, logo o exemplo respeita o fator de aproximação do algoritmo.

2.5 Cobertura por Conjuntos (*Set Cover*)

A melhor forma de introduzir o Problema da Cobertura por Conjuntos é apresentar primeiro o que vem a ser uma cobertura.

Definição 2.5 (Cobertura)**.** *Dado um conjunto finito E e uma coleção $\mathcal{S}$ de subconjuntos de E, dizemos que uma subcoleção $T \subseteq \mathcal{S}$ é* **cobertura** *de E em $\mathcal{S}$ se todo elemento de E pertence a algum conjunto de T.*

Quando ficar evidente qual é a coleção de conjuntos que a cobertura de E deriva, vamos só falar que T é cobertura de E.

Dada a definição de cobertura, veja que dados E e $\mathcal{S}$, podem existir várias subcoleções de $\mathcal{S}$ que sejam coberturas de E. Assim, também temos uma função c que atribui um custo para cada conjunto pertencente a $\mathcal{S}$, de forma que podemos comparar as possíveis coberturas para E nos baseando em seus custos, sendo que o que desejamos é encontrar a cobertura com menor custo.

O parágrafo acima descreve de maneira informal o Problema da Cobertura por Conjuntos. Segue abaixo sua descrição formal.

Problema 2.10 (Cobertura por Conjuntos)**.** *Dados um conjunto finito E, uma coleção $\mathcal{S}$ de suconjuntos de E e uma função de custo $c : \mathcal{S} \to \mathbb{Q}^+$, desejamos encontrar uma cobertura T de E tal que $c(T) = \sum_{C \in T} c(C)$ seja mínimo.*

Perceba que o problema é de minimização e que uma instância é definida pelo conjunto E, a coleção $\mathcal{S}$ e a função c. Toda solução do problema é uma cobertura de E, sendo que o custo de cada solução é definido pela custo da cobertura.

O Problema 2.10 pertence a $\mathcal{NP}$-difícil [2]. Antes de falarmos sobre o algoritmo de aproximação que apresentaremos nessa seção, devemos apresentar o que é o número harmônico.

Definição 2.6 (Número Harmônico)**.** *Dado $n \in \mathbb{N}$, definimos o* **número harmônico** H_n *da seguinte maneira:*

$$H_n = \sum_{k=1}^{n} \frac{1}{k}\,.$$

O algoritmo de aproximação que apresentaremos chama-se algoritmo de Chvátal [2] e é uma H_n-aproximação para a Cobertura por Conjuntos, onde $n = |E|$. Ele é apresentado no Algoritmo 7.

Algoritmo 7: COBERTURA-CHVÁTAL(E, $\mathcal{S}$, c)

Entrada: Conjunto E, coleção de conjuntos finitos $\mathcal{S}$ e função de custo nos conjuntos de $\mathcal{S}$
Saída: Cobertura de E

se $E == \emptyset$ **então**
  retorna $\emptyset$
fim
senão
  Seja $Z \in \mathcal{S}$ tal que $c(Z)/|Z \cap E|$ é mínimo
  $E' = E \setminus Z$
  $\mathcal{S}' = \{S \in \mathcal{S} : S \cap E' \neq \emptyset\}$
  seja c' a restrição de c apenas aos conjuntos em $\mathcal{S}'$
  $T =$ COBERTURA-CHVÁTAL(E', $\mathcal{S}'$, c')
  $T = T \cup \{Z\}$
  retorna T
fim

Perceba que o algoritmo é recursivo e baseia-se numa estratégia gulosa para construir a cobertura para E. Quando o algoritmo decide qual conjunto contido em $\mathcal{S}$ será escolhido, chamado de Z, é feita uma escolha localmente ótima. Essa escolha é baseada no valor de $c(Z)/|Z \cap E|$, que indica quanto o conjunto custa por elemento ainda não coberto em E. É justamente a escolha de Z que caracteriza COBERTURA-CHVÁTAL como algoritmo guloso.

Embora o algoritmo seja recursivo é fácil identificar que, no máximo,

serão feitas $|E|$ chamadas recursivas, devido à condição de parada na linha 2 e a forma como a cada E' é definido na linha 6.

Veja que na estrutura **se** as operações realizadas consomem tempo $O(1)$. Já na estrutura **senão**, as linhas 5, 7 e 8 são dependentes da quantidade de conjuntos em $\mathcal{S}$, ou seja, possuem complexidade $O(|\mathcal{S}|)$. Ainda na estrutura **senão**, temos as linhas 6, 10 e 11 com custo $O(1)$.

A parte complicada em analisar a complexidade do algoritmo é a recursão. Veja que para cada chamada recursiva o tempo dominante é $|\mathcal{S}|$ e a cada nova chamada a coleção que faz parte da instância tem seu tamanho decrementado por pelo menos 1, afinal retiramos Z dessa coleção. Como não sabemos quantas chamadas serão necessárias para que todos os elementos de E sejam cobertos e, na pior das hipóteses, cada chamada só cobrirá um elemento de E, a complexidade do algoritmo torna-se $O(|S|\,|E|)$.

Segue um lema que estabelece uma relação entre o custo da solução ótima e a expressão usada na linha 5 para seleção do conjunto Z. Esse lema será usado para demonstrar o fator de aproximação do algoritmo.

Lema 2.7. *Dada uma instância $\langle E, \mathcal{S}, c\rangle$ para o Problema da Cobertura por Conjuntos e dado o algoritmo* Cobertura-Chvátal*, vale que*

$$\frac{c(Z)}{|Z \cap E|}|E| \leq OPT_{CC}(E, \mathcal{S}, c)\,.$$

Demonstração. Veja que $\forall\ Z' \in \mathcal{S} \setminus \{Z\}$ vale que

$$\frac{c(Z)}{|Z \cap E|} \leq \frac{c(Z')}{|Z' \cap E|}\,,$$

o que implica que para qualquer que seja T,

$$\frac{c(Z)}{|Z \cap E|}|E| \leq \sum_{S \in T} \frac{c(S)}{|S \cap E|}|S \cap E| = \sum_{S \in T} c(S) = c(T).$$

Perceba que uma solução ótima é um caso específico de T, logo

$$\frac{c(Z)}{|Z \cap E|}|E| \leq OPT_{CC}(E, \mathcal{S}, c).$$

□

Segue o teorema referente ao fator de aproximação do algoritmo.

Teorema 2.8. *Seja $(E, \mathcal{S}, c)$ uma instância da Cobertura por Conjuntos. Se $n = |E|$, então o algoritmo* Cobertura-Chvátal *é uma H_n aproximação para o problema.*

Demonstração. Vamos realizar essa demonstração por indução em n, ou seja, em $|E|$.

Base: Se $n = 0$, então o algoritmo devolve $\emptyset$. Assim, a cobertura é ótima, pois é a única possível.

Hipótese: Sejam $|E'| = n - k > 0$, onde $k \in \mathbb{N}$, e T' a cobertura de E' devolvida na linha 9. Então vale que $c'(T') \leq H_{n-k}\, OPT_{CC}(E', \mathcal{S}', c')$.

Indução: Seja $|E| = n$. Perceba que $E' \subseteq E$, logo

$$OPT_{CC}(E', \mathcal{S}', c') \leq OPT_{CC}(E, \mathcal{S}, c)\,. \tag{2.15}$$

Perceba também que devido à desigualdade do Lema 2.7,

$$c(T) = c(Z) + c'(T') \leq \frac{|Z \cap E|}{|E|}\, OPT_{CC}(E, \mathcal{S}, c) + c'(T')\,. \tag{2.16}$$

Agora, usando da hipótese e da desigualdade (2.16) concluímos que

$$c(T) \leq \frac{|Z \cap E|}{|E|}\, OPT_{CC}(E, \mathcal{S}, c) + H_{n-k}\, OPT_{CC}(E', \mathcal{S}', c')\,. \tag{2.17}$$

Unindo as desigualdades (2.15) e (2.17) obtemos

$$c(T) \leq \frac{k}{n}\, OPT_{CC}(E, \mathcal{S}, c) + H_{n-k}\, OPT_{CC}(E, \mathcal{S}, c)\,. \tag{2.18}$$

Lembrando que da hipótese vem que $k = |Z \cap E|$ e que $n = |E|$.

Sabemos que as seguintes igualdades são validas:

- $\frac{k}{n} = \sum_{i=0}^{k-1} \frac{1}{n-i}$;
- $H_{n-k} = \sum_{i=1}^{n-k} \frac{1}{i}$.

Assim, a seguinte expressão é verdadeira:

$$\frac{k}{n} + H_{n-k} = \frac{1}{n} + \cdots + \frac{1}{n-k+1} + \frac{1}{n-k} + \cdots + 1 = H_n \,. \tag{2.19}$$

Unindo os resultados de (2.18) e (2.19) obtemos

$$c(T) \leq H_n \, OPT_{CC}(E, \mathcal{S}, c) \,.$$

□

Segue um exemplo de execução do algoritmo que mostra que o fator de aproximação encontrado é justo.

Exemplo 2.7. *Suponha que desejemos cobrir um conjunto E com n elementos, sendo que a coleção $\mathcal{S}$ que temos disponível para cobri-lo possui n conjuntos unitários, cada um com um dos n elementos de E, e um outro conjunto com todos os n elementos de E.*

A função c dessa instância é definida da seguinte forma:

- *Cada um dos n conjuntos unitários possui uma relação bijetiva de custo com o conjunto $\left\{\frac{1+\epsilon}{n}, \frac{1+\epsilon}{n-1}, \ldots, 1+\epsilon\right\}$;*
- *O único conjunto não unitário possui custo $1+\epsilon$.*

Considere que ϵ é um valor infinitesimal.

Perceba que na primeira chamada do algoritmo podem ser escolhidos como Z o grande conjunto com todos os n elementos ou o conjunto unitário com custo $\frac{1+\epsilon}{n}$. Isso é possível pois ambos têm a mesma relação de custo-cobertura. Assuma então que o conjunto escolhido pelo algoritmo foi o unitário.

Veja que a cada chamada recursiva existe um conjunto unitário de mesmo custo-cobertura que o grande conjunto. Suponha que o conjunto unitário sempre seja o escolhido.

Veja que ao final obteremos uma cobertura de custo

$$\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + 1\right)(1+\epsilon) = H_n(1+\epsilon).$$

Assim, a solução devolvida possui custo $H_n(1+\epsilon)$, sendo que a solução ótima só custaria $1+\epsilon$, que seria a escolha do maior conjunto.

2.6 Supercadeia Mínima (*Shortest Superstring*)

Considere as duas cadeias de caracteres (*strings*) a seguir:

$$S_1 = (a, t, c, g, g, a, c) \quad S_2 = (g, g, a, c, t, t, g)$$

As seguintes duas cadeias diferentes contêm ambas S_1 e S_2:

$$SC_1 = (a, t, c, g, g, a, c, t, t, g) \quad SC_2 = (g, g, a, c, t, t, g, a, t, c, g, g, a, c)$$

Chamamos SC_1 e SC_2 de ***supercadeias*** (*superstrings*) de S_1 e S_2, pois ambas contêm essas duas cadeias menores.

Tanto SC_1 quanto SC_2 contêm S_1 e S_2, porém, SC_1 possui 4 caracteres a menos que SC_2. Assim, dentre as supercadeias possíveis que contêm S_1 e S_2, vamos chamar SC_1 de ***supercadeia mínima***.

Basicamente o que fizemos acima foi dar um exemplo do problema da supercadeia mínima. Abaixo segue a descrição formal desse problema.

Problema 2.11 (Supercadeia Mínima)**.** *Dado um conjunto finito* $\mathcal{S} = \{S_1, \ldots, S_n\}$ *com* n *cadeias, desejamos encontrar uma supercadeia com o menor número de caracteres que contenha todas as cadeias em* $\mathcal{S}$*.*

Sem perda de generalidade vamos assumir que em uma instância $\langle \mathcal{S} \rangle$ do problema, nenhuma cadeia de $\mathcal{S}$ é subcadeia da outra. Note que o tamanho do problema é dado por $\sum_{i=1}^{n} |S_i|$. O problema da supercadeia mínima pertence a $\mathcal{NP}$-difícil [6].

Podemos perceber que o Problema 2.11 é de minimização e possui como entrada um conjunto finito $\mathcal{S}$ de cadeias. Cada solução viável para uma determinada instância é uma supercadeia. O valor de cada solução viável é definido pela sua quantidade de caracteres.

Ao todo apresentaremos três algoritmos nessa seção, cada um com um fator de aproximação diferente para o problema.

O primeiro algoritmo que iremos apresentar é conjecturado como uma 2-aproximação para o problema da Supercadeia Mínima [6], porém, esse fator de aproximação até agora foi demonstrado. Antes de apresentarmos o algoritmo vamos definir o que é uma *sobreposição* no contexto desse problema.

Definição 2.7 (Sobreposição)**.** *Dadas duas cadeias S e T, uma* **sobreposição** *entre elas é um valor qualquer de caracteres em comum entre um sufixo de S e um prefixo de T.*

Nas cadeias S_1 e S_2 do exemplo inicial a sobreposição máxima é igual a 4.

O Algoritmo 8 formaliza a ideia do primeiro algoritmo que veremos.

Algoritmo 8: ALG-SC($\mathcal{S}$)

Entrada: Conjunto $\mathcal{S}$ com n cadeias
Saída: Supercadeia que contém todas as cadeias em $\mathcal{S}$

```
1 T = S
2 enquanto |T| > 1 faça
3 |   Selecione duas cadeias S_i, S_j ∈ T com maior sobreposição
4 |   S_ij = SOBREPÕE(S_i, S_j) /* Menor supercadeia de S_i e S_j */
5 |   T = T \ {S_i, S_j}
6 |   T = T ∪ {S_ij}
7 fim
8 retorna a cadeia contida em T
```

O Algoritmo 8 recebe como entrada um conjunto finito $\mathcal{S}$ de cadeias. Inicialmente, T é uma cópia de $\mathcal{S}$. A cada iteração do laço **enquanto**, selecionamos as cadeias S_i e S_j em T com maior sobreposição. A função SOBREPÕE devolve a menor supercadeia que contenha as duas cadeias dadas como argumentos. Substituímos S_i e S_j em T pela supercadeia devolvida pela função SOBREPÕE. Veja que ao final de cada iteração diminuímos em 1 a cardinalidade de T. Esse processo vai repetindo-se até que $|T| = 1$, ou seja, quando T só possuir uma cadeia que na realidade é uma supercadeia com todos os elementos de $\mathcal{S}$.

Veja que Alg-SC é um algoritmo guloso, sendo que a escolha gulosa é realizada na linha 3. Essa escolha é considerada gulosa, pois só importa-se em encontrar a supercadeia mínima entre dois elementos de T no momento da seleção.

Como já dito, não há demonstração para o fator de aproximação do algoritmo [6]. Segue um exemplo de instância para esse algoritmo que mostra que o fator não pode ser menor do que 2.

Exemplo 2.8. *Considere que as seguintes cadeias compõem uma entrada $\mathcal{S}$ para o Algoritmo 8, em que k é um natural não-nulo:*

$$S_1 = (a, \underbrace{b, \ldots, b}_{k})$$
$$S_2 = (\underbrace{b, \ldots, b}_{k}, c)$$
$$S_3 = (\underbrace{b, \ldots, b}_{k+1})$$

Veja que na linha 3 poderiam ser selecionadas quaisquer duas cadeias. Suponha que as cadeias selecionadas primeiro sejam S_1 e S_2. A supercadeia formada seria $S_{12} = (a, \underbrace{b, \ldots, b}_{k}, c)$.

Assim, numa segunda iteração as cadeias disponíveis para serem sobrepostas seriam S_3 e S_{12}. Ao final da execução a supercadeia formada seria $S_{123} = (a, \underbrace{b, \ldots, b}_{k}, c, \underbrace{b, \ldots, b}_{k+1})$, solução essa que contém $2k + 3$ caracteres.

No entanto, a solução ótima seria a supercadeia $(a, \underbrace{b, \ldots, b}_{k+1}, c)$, que possui $k + 3$ caracteres.

A solução obtida possui assintoticamente duas vezes a quantidade de caracteres da solução ótima.

Outro algoritmo para a Supercadeia Mínima surge de uma redução para a Cobertura por Conjuntos (Problema 2.10). O fator de aproximação dessa

redução é $2\,H_n$ se usarmos o algoritmo apresentado na Seção 2.5, onde $n = |E| = |\mathcal{S}|$.

Seja $\mathcal{S} = \{S_1, \ldots, S_n\}$ uma instância do problema da supercadeia mínima. Lembrando que uma instância do problema da Cobertura por Conjuntos é composta por:

- Conjunto E de elementos que desejamos cobrir;
- Coleção $\mathcal{Z}$ de subconjuntos de E; e
- Função $c : \mathcal{Z} \to \mathbb{R}^+$ de custo.

Considere a notação:

1. s_i e s_j são cadeias de $\mathcal{S}$, sendo que $i \neq j$;
2. k é um inteiro positivo; e
3. π_{ijk} é a cadeia formada pela sobreposição de k símbolos entre s_i e s_j.

Também definimos um conjunto Π como sendo o conjunto de todas as cadeias π_{ijk} que podem ser construídas a partir de quaisquer duas cadeias s_i e s_j em $\mathcal{S}$ e quaisquer valores possíveis de k.

Dada uma cadeia w qualquer, definimos uma relação de w em $\mathcal{S}$ por meio do conjunto $sub(w) = \{s_i \in \mathcal{S} \mid s_i \text{ é subcadeia de } w\}$.

Finalmente, dada a instância $\langle \mathcal{S} \rangle$ para a Supercadeia Mínima, construímos uma instância $\langle E, \mathcal{Z}, c \rangle$ para a Cobertura por Conjuntos em que:

- $E = \mathcal{S}$;
- $\mathcal{Z} = \mathcal{S} \cup \Pi$;
- c é definida de forma que para cada $s \in \mathcal{Z}$, $c(s) = |s|$.

O algoritmo com fator de aproximação $2\,H_n$ para Supercadeia Mínima é dado no Algoritmo 9.

Algoritmo 9: $2H_n$-Aprox-SC($\mathcal{S}$)

Entrada: Conjunto finito de cadeias $\mathcal{S}$
Saída: Supercadeia com todas as cadeias em $\mathcal{S}$

```
1 Seja ⟨E, Z, c⟩ construída a partir de ⟨S⟩
2 C = Cobertura-Chvátal(E, Z, c) /* C é uma cobertura para E   */
3 Seja C = {c_1, c_2, ..., c_m} /* Cada c_i é uma cadeia        */
4 G = Concatena(C) /* Concatenação de todos os elementos de C  */
5 retorna G
```

Na linha 2, o Algoritmo 7, apresentado na Seção 2.5 como H_n-aproximação para Cobertura por Conjuntos, é usado. Como já sabemos, tal algoritmo é polinomial. A operação de concatenação que ocorre na linha 4 ocorre em tempo $O(|E|) = O(|\mathcal{S}|)$ se a implementação de concatenação de strings for realizada em tempo constante (o que pode ser feito com uma representação em listas, por exemplo). Concluindo, de fato o Algoritmo 9 é polinomial.

Note que a cobertura C devolvida cobre o conjunto $E = \mathcal{S}$, o que garante que a supercadeia G devolvida com certeza possui todos os elementos de $\mathcal{S}$, sendo assim uma supercadeia de $\mathcal{S}$.

Sejam OPT_{CC} o custo da solução ótima para a Cobertura por Conjuntos e OPT_{SM} o custo da solução ótima para a Supercadeia Mínima. Os lemas a seguir serão usados na demonstração do fator de aproximação do Algoritmo 9. Ele mostra uma relação entre os valores OPT_{CC} e OPT_{SM}.

Lema 2.8. *Seja $\langle E, \mathcal{Z}, c\rangle$ uma instância da Cobertura por Conjuntos adaptada de $\langle \mathcal{S}\rangle$, uma instância de Supercadeia Mínima. Então*

$$OPT_{SM}(\mathcal{S}) \leq OPT_{CC}(E, \mathcal{Z}, c)\,.$$

Demonstração. Seja $A \subseteq \mathcal{Z}$ uma cobertura por conjuntos para $\langle E, \mathcal{Z}, c\rangle$ ótima, isto é, $c(A) = OPT_{CC}(E, \mathcal{Z}, c)$.

Veja que independente da cobertura C devolvida por Cobertura-Chvá-

TAL, vale que

$$c(A) \leq c(C).$$

Concatene todas as cadeias de A em uma supercadeia B. Veja que o custo de A é igual ao tamanho de B, ou seja,

$$c(A) = |B|\,.$$

Como B é um caso particular de supercadeia para $\mathcal{S}$, então $OPT_{SM}(\mathcal{S}) \leq |B|$, o que nos leva a

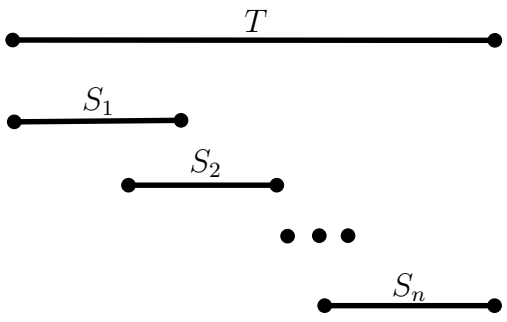
$$OPT_{SM}(\mathcal{S}) \leq |B| = c(A) = OPT_{CC}(E, \mathcal{Z}, c)\,.$$

□

Lema 2.9. *Seja $\langle E, \mathcal{Z}, c\rangle$ uma instância da Cobertura por Conjuntos adaptada de $\langle\mathcal{S}\rangle$, uma instância de Supercadeia Mínima. Então*

$$OPT_{CC}(E, \mathcal{Z}, c) \leq 2 \cdot OPT_{SM}(\mathcal{S})\,.$$

Demonstração. Dada uma supercadeia T que seja solução ótima para Supercadeia Mínima sobre $\mathcal{S}$, considere $Seq = (S_1, S_2, \ldots, S_n)$ a sequência das cadeias de $\mathcal{S}$ na ordem em que elas aparecem em T.

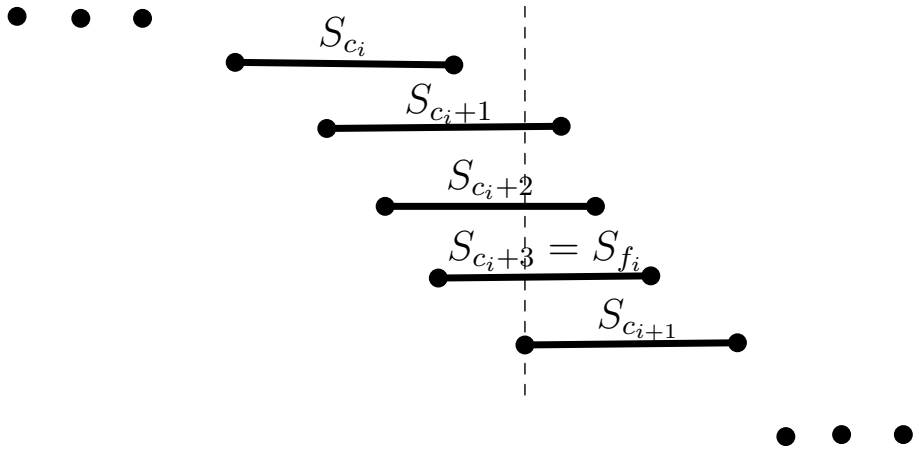


Como nenhuma cadeia é subcadeia de outra cadeia em $\mathcal{S}$ a ordem que as cadeias S_i aparecem em Seq também indica a ordem em que as cadeias

terminam.

Vamos particionar Seq em h grupos de cadeias contínuas no próprio Seq. Cada um dos h grupos vai ser representado por H_i, onde $i \in \{1, 2, \ldots, h\}$ e indica a ordem dos grupos em Seq. Vamos denotar por c_i e f_i, respectivamente, os índices em Seq da primeira e da última cadeias do grupo H_i, onde $i \in \{1, 2, \ldots, h\}$.

A forma como dividimos os grupos segue a seguinte regra: se S_{c_i} é a primeira cadeia do grupo H_i, então S_{f_i} é a última cadeia em Seq a ter sobreposição não nula com S_{c_i}.



Agora, para todo par (c_i, f_i), vamos definir $k_i > 0$ como o valor da sobreposição entre S_{c_i} e S_{f_i} na supercadeia T. É importante notar que o valor de k_i pode não ser igual à sobreposição máxima entre tais cadeias.

Veja que a supercadeia formada pela concatenação de todas as cadeias $\pi_{c_i,f_i,k_i} = \phi_i$ é uma solução para o problema da Supercadeia Mínima na instância $\langle \mathcal{S} \rangle$.

Além disso, o conjunto Φ que reúne todas as cadeias ϕ_i é uma solução para o problema da Cobertura por Conjuntos em $\langle E, \mathcal{Z}, c \rangle$. Assim, vale a desigualdade

$$OPT_{CC}(E, \mathcal{Z}, c) \leq c(\Phi)\,. \tag{2.20}$$

Da forma que as cadeias ϕ_i foram definidas, nenhuma cadeia ϕ_i pode ter sobreposição diferente de 0 com a cadeia ϕ_{i+2}. Isso é verdade porque caso houvesse tal sobreposição a cadeia ϕ_{i+1} seria subcadeia de ϕ_i e ϕ_{i+2}.

É possível ver que a concatenação das cadeias em Φ gera uma supercadeia S_Φ que é solução viável para o problema da Supercadeia Mínima em $\langle \mathcal{S} \rangle$. Pela discussão anterior, em S_Φ cada caractere não é repetido mais que duas vezes em comparação à solução ótima, ou seja,

$$c(S_\Phi) \leq 2 \cdot OPT_{SM}(\mathcal{S}) \,. \tag{2.21}$$

Assim, unindo as equações (2.20) e (2.21) obtemos que

$$OPT_{CC}(E, \mathcal{Z}, c) \leq 2 \cdot OPT_{SM}(\mathcal{S}) \,. \tag{2.22}$$

□

O teorema a seguir demonstra o fator de aproximação $2\,H_n$ do Algoritmo 9.

Teorema 2.9. *O algoritmo* 2Hn-Aprox-SC *é uma* $(2\,H_n)$*-aproximação para a Supercadeia Mínima, sendo* n *a quantidade de cadeias na instância.*

Demonstração. Seja $\mathcal{S}$ uma instância da Supercadeia Mínima, onde $|\mathcal{S}| = n$, e seja $(E, \mathcal{Z}, c)$ uma instância adaptada de $\mathcal{S}$ para a Cobertura por conjuntos.

Seja G a solução devolvida por 2Hn-Aprox-SC. O algoritmo usado para encontrar a cobertura C de $(E, \mathcal{Z}, c)$ é uma H_n-aproximação, de forma que

$$|G| = c(C) \leq H_n \cdot OPT_{CC}(E, \mathcal{Z}, c) \,.$$

Pelo Lema 2.9, temos $OPT_{CC}(E, \mathcal{Z}, c) \leq 2\,OPT_{SM}(\mathcal{S})$.

Unindo as duas inequações, temos

$$|G| \leq 2\,H_n\,OPT_{SM}(\mathcal{S}) \,.$$

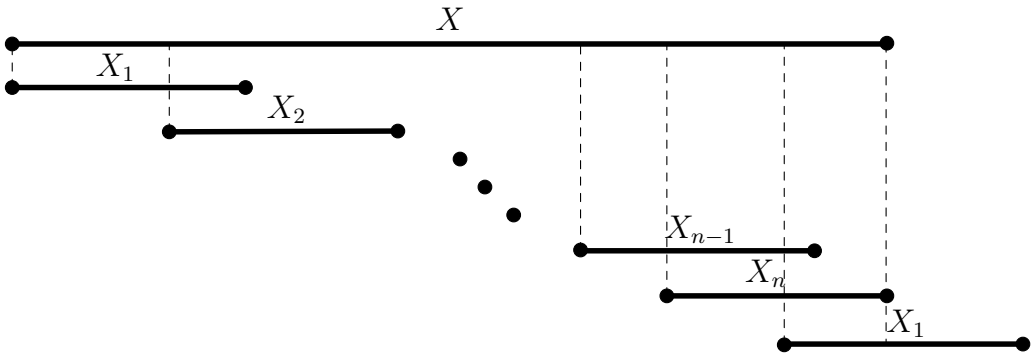


Figura 2.1: Ilustração de uma solução X para $\mathcal{S}$ e a renomeação das strings de $\mathcal{S}$ de acordo com sua ordem de aparição em X.

□

Provado o fator de aproximação do Algoritmo 9, que usa uma redução da Cobertura por Conjuntos, vamos passar para o último algoritmo de aproximação para Supercadeia Mínima que iremos apresentar. Trata-se de uma 4-aproximação para o problema que também usa uma redução entre problemas. Nesse caso, faremos uma redução da Supercadeia Mínima para o problema conhecido como Cobertura por Ciclos. Vamos começar estabelecendo algumas notações que serão usadas.

Dadas duas cadeias S_i e S_j vamos denotar por:

- $Sobre(S_i, S_j)$ a cadeia formada pelos caracteres que sobrepõem-se em uma supercadeia miníma entre S_i e S_j, onde S_i fornece o sufixo dessa sobreposição e S_j fornece o prefixo;

- $Pref(S_i, S_j)$ a cadeia obtida de S_i a partir da remoção de $Sobre(S_i, S_j)$ de seu sufixo.

Para um melhor entendimento dessas notações, considere as cadeias $C_1 = (a, b, c, c, c, b, b)$ e $C_2 = (b, b, c, b, a, a, a)$. Aqui, $Sobre(C_1, C_2) = (b, b)$ e $Pref(C_1, C_2) = (a, b, c, c, c)$.

Seja X uma solução para alguma instância $\mathcal{S}$ do problema da Supercadeia Mínima. Vamos renomear as cadeias de $\mathcal{S}$ com respeito à ordem em que elas aparecem em X. Assim, $Seq_X = \{X_1, X_2, \ldots, X_n\}$ e X_i é a cadeia de $\mathcal{S}$ tal que outras $i-1$ cadeias de $\mathcal{S}$ aparecem antes de X_i em X. Veja a Figura 2.1 para um exemplo dessa discussão.

Assim, por construção, vale a seguinte igualdade:

$$c(X) = |Pref(S_1, S_2)| + |Pref(S_2, S_3)| + \cdots + |Pref(S_n, S_1)| + |Sobre(S_n, S_1)|\,.$$

Em particular, essa igualdade pode ser aplicada a uma supercadeia S_{OPT} que seja solução ótima do problema. Se a sequência de cadeias para tal solução é $Seq_{OPT} = \{S'_1, S'_2, \ldots, S'_n\}$, então

$$OPT_{SM}(\mathcal{S}) = |Pref(S'_1, S'_2)| + \cdots + |Pref(S'_n, S'_1)| + |Sobre(S'_n, S'_1)|\,.$$

Com essa expressão para $OPT_{SM}(\mathcal{S})$, podemos criar uma relação entre encontrar uma Supercadeia Mínima em $\mathcal{S}$ e resolver o Caixeiro Viajante em um digrafo $G_{\mathcal{S}}$ ponderado nas arestas construído da seguinte forma:

- Cada uma das cadeias de $\mathcal{S}$ é um vértice do digrafo, sendo que nomearemos cada vértice com seu índice em Seq_{OPT};
- Para cada par de vértices i, j de $V(G_{\mathcal{S}})$, tal que se $i \neq j$, então estabelecemos uma aresta ij orientada que parte de i em direção a j;
- Para cada aresta orientada ij, o custo de tal aresta é $|Pref(S'_i, S'_j)|$.

Tal grafo $G_{\mathcal{S}}$ é chamado de ***grafo de prefixos de*** $\mathcal{S}$.

Veja que $|Pref(S'_1, S'_2)| + |Pref(S'_2, S'_3)| + \cdots + |Pref(S'_n, S'_1)|$ é o custo de um ciclo hamiltoniano com menor custo em $G_{\mathcal{S}}$, ou seja, é uma solução ótima do Caixeiro Viajante em $G_{\mathcal{S}}$[3]. Assim, concluímos que a solução ótima

[3]Por mais que $G_{\mathcal{S}}$ seja um grafo orientado, o enunciado do Caixeiro Viajante é análogo ao Problema 2.8.

do Caixeiro Viajante para $G_{\mathcal{S}}$ é um limitante inferior para a solução ótima da Supercadeia Mínima em $\mathcal{S}$. Essa informação não tem muita utilidade prática para nós, afinal, Caixeiro Viajante pertence a $\mathcal{NP}$-difícil.

Um outro limitante inferior para a Supercadeia Mínima deriva do problema da Cobertura por Ciclos. Segue o enunciado de tal problema.

Problema 2.12 (Cobertura por Ciclos)**.** *Dado um grafo G e uma função de custo $f : E(G) \to \mathbb{R}^+$, desejamos encontrar um conjunto $\mathcal{C}$ de ciclos disjuntos em G, tal que todo vértice de $V(G)$ está presente em algum ciclo de $\mathcal{C}$ e $f(\mathcal{C})$ é mínimo.*

Uma instância da Cobertura por Ciclos é constituída de um grafo e uma função de peso nas arestas desse grafo. Podemos perceber que trata-se de um problema de minimização.

Então, a ideia é usar uma Cobertura por Ciclos no grafo de prefixos de $\mathcal{S}$ como limitante inferior para OPT_{SM}. O algoritmo que apresentaremos adapta uma instância da Supercadeia Mínima para uma instância da Cobertura por Ciclos. A resposta devolvida para Cobertura por Ciclos será usada na construção de uma solução viável da Supercadeia Mínima com fator de aproximação 4.

Observe que a solução ótima do Caixeiro Viajante em $G_{\mathcal{S}}$ é $(1, 2, \ldots, n, 1)$, sendo que essa sequência também representa uma cobertura por ciclos. Assim, a solução ótima para Cobertura por Ciclos com certeza é limitante inferior da Supercadeia Mínima.

Diferente do Caixeiro Viajante, podemos calcular uma Cobertura por Ciclos mínima em tempo polinomial [6]. Uma forma de encontrar tal cobertura mínima vem da relação desse problema com emparelhamentos perfeitos. Toda cobertura por ciclos em $G_{\mathcal{S}}$ corresponde a um emparelhamento perfeito (Definição 2.4) em um grafo H bipartido, ou seja, podemos reduzir o problema da Cobertura por Ciclos para o problema de encontrar um emparelhamento perfeito de custo mínimo em H. A construção do grafo bipartido H, onde $V(H) = (U, V)$ para esse problema é feita da seguinte forma:

- $U = \{u_1, u_2, \ldots, u_n\}$ e $V = \{v_1, v_2, \ldots, v_n\}$, lembrando que $|\mathcal{S}| = n$;

- Para cada par de vértices v_i e u_j, tal que $i \neq j$, temos uma aresta $v_i u_j$ com custo $|Pref(S_i, S_j)|$.

Vamos agora apresentar algumas notações e nomenclaturas que serão usadas. Dado um ciclo $C = (a_1, a_2, \ldots, a_k)$ no grafo $G_{\mathcal{S}}$, onde $k \leq n$,

- $\alpha(C)$ é uma cadeia, que é resultante da concatenação de $Pref(S_{a_1}, S_{a_2})$, $Pref(S_{a_2}, S_{a_3})$, ..., $Pref(S_{a_{k-1}}, S_{a_k})$ e $Pref(S_{a_k}, S_{a_1})$. Assim, $|\alpha(C)|$ é o peso do ciclo C;

- $\phi(C)$ é uma cadeia, que é a concatenação de $\alpha(C)$ com alguma cadeia S_{a_i}, onde S_{a_i} é a *cadeia representativa* de C.

Veja que a cadeia $\phi(C)$ acima depende de qual é a cadeia representativa do ciclo, que pode ser qualquer uma das cadeias de C. Por ser cíclico, qualquer outro vértice do ciclo pode ocupar a posição a_1. Por isso, vamos sempre considerar que em a_1 está o índice da cadeia representativa de $C = (a_1, \ldots, a_k)$.

Perceba também que $\phi(c)$ é uma supercadeia das cadeias $S_{a_1}, S_{a_2}, \ldots, S_{a_k}$.

Passadas essas notações, temos o Algoritmo 10, que é uma 4-aproximação para a Supercadeia Mínima.

Algoritmo 10: 4-Aprox-SC($\mathcal{S}$)

Entrada: Conjunto finito de cadeias $\mathcal{S}$
Saída: Supercadeia que contém todas as cadeias em $\mathcal{S}$

```
1 Seja G o grafo de prefixos de S
2 Seja f a função de custo das arestas em E(G)
3 C = Cob-Ciclos(G, f) /* Encontra cobertura por ciclos mínima em G*/
4 Seja C = {C_1, C_2, ..., C_m} /* Cada C_i é um ciclo da cobertura */
5 Seja Φ o conjunto das cadeias φ(C_i) para todo C_i ∈ C
6 A = Concatena(Φ) /* Concatenação de todas as cadeias em Φ */
7 retorna A
```

É fácil perceber que a cadeia A devolvida pelo algoritmo é supercadeia de todas as cadeias em $\mathcal{S}$, afinal temos a garantia de que a cobertura por ciclos incluiu toda cadeia de $\mathcal{S}$ em um ciclo de $\mathcal{C}$.

Seja $\mathcal{C} = \{C_1, \ldots, C_m\}$ o conjunto de ciclos da cobertura encontrada no algoritmo 4-Aprox-SC. Vamos escrever $C_i = (a_{i1}, \ldots, a_{ik})$, em que $S_{a_{i1}}$ é a cadeia representativa de C_i.

Sabemos que a solução ótima da Cobertura por Ciclos no grafo de prefixos de $\mathcal{S}$ é limitante inferior para a solução ótima da Supercadeia Mínima. No Algoritmo 10 encontramos uma cobertura $\mathcal{C} = \{C_1, \ldots, C_m\}$ por ciclos mínima no grafo de prefixos de $\mathcal{S}$, sendo que o custo dessa cobertura é dado por $\sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)|$, devido à maneira como o custo das aresta do grafo de prefixos é definido. Como estamos falando da solução ótima para Cobertura por Ciclos no grafo de prefixos de $\mathcal{S}$, então

$$\sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)| = OPT_{CobCic}(G_{\mathcal{S}}) \leq OPT_{SM}(\mathcal{S}) \tag{2.23}$$

Perceba que, se para cada ciclo $C_i \in \mathcal{C}$ conseguirmos encontrar uma cadeia representativa de tamanho menor que o peso de C_i, então

$$|A| = \sum_{i=1}^{|\Phi|} |\phi(C_i)| = \sum_{i=1}^{|\Phi|} |\alpha(C_i)| + |S_{a_{i1}}| \leq \sum_{i=1}^{|\Phi|} |\alpha(C_i)| + |\alpha(C_i)| \leq 2OPT_{SM}(\mathcal{S}),$$

onde a última desigualdade segue de 2.23, lembrando que $|\mathcal{C}| = |\Phi|$. Ou seja, a cadeia devolvida pelo algoritmo terá peso no máximo $2OPT_{SM}(\mathcal{S})$.

Agora, caso todo C só possua cadeias longas, no sentido de todas as cadeias possuírem tamanho maior que o peso do ciclo, então a cadeia representativa também será longa e é por causa desse caso que o algoritmo não é uma 2-aproximação e sim uma 4-aproximação.

Seguem dois lemas, ambos com objetivo auxiliar na demonstração do fator de aproximação caso só tenhamos cadeias representativas longas. O primeiro

lema será usado na demonstração do segundo, que por sua vez garante um limite no tamanho das cadeias representativas dos ciclos. Assim, ele será usado diretamente na demonstração do fator de aproximação. Em ambos os lemas vamos denotar por T^∞ a concatenação de infinitas cadeias T.

Lema 2.10. *Dados uma cadeia T qualquer e um conjunto de cadeias $\mathcal{S}$ que é instância para Supercadeia Mínima, se cada cadeia de $\mathcal{S}' \subseteq \mathcal{S}$ é subcadeia de T^∞, então existe um ciclo C de custo no máximo $|T|$ no grafo de prefixos de $\mathcal{S}$ que cobre todos os vértices correspondentes às cadeias contidas em $\mathcal{S}'$.*

Demonstração. Vamos indexar as cadeias de $\mathcal{S}'$ com base na ordem em que cada uma dessas cadeias tem sua primeira ocorrência em T^∞. Assim, $Seq_{T^\infty} = \{S_{t1}, \ldots, S_{t|\mathcal{S}'|}\}$. Claramente, o início de cada uma das cadeias é em um ponto distinto de T^∞ e estará presente na primeira cópia de T.

Seja $G_\mathcal{S}$ o grafo de prefixos de $\mathcal{S}$ e considere o ciclo $C = (t1, \ldots, t|\mathcal{S}'|, t1)$ construído a partir de Seq_{T^∞}. É evidente que o custo de C é no máximo $|T|$. □

Segue o segundo lema, como já dito, tal lema mostra um limite para a sobreposição de cadeias representativas entre dois ciclos de uma cobertura por ciclos, isso é importante, pois está relacionado ao quanto de custo a mais temos na concatenação final das cadeias ϕ no Algoritmo 10. Esse lema nos apresenta uma relação entre a sobreposição das cadeias representativas de duas cadeias e o tamanho de das cadeias de concatenações das mesmas cadeias.

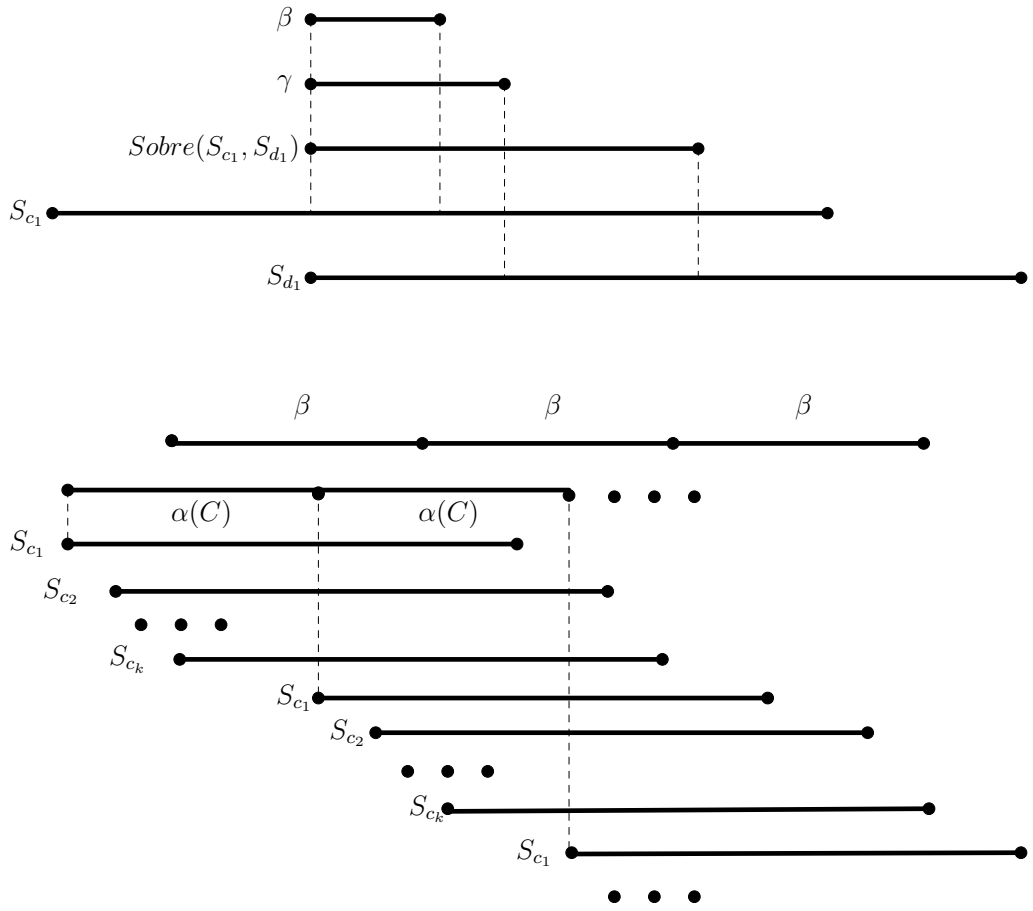
Lema 2.11. *Sejam um conjunto de cadeias $\mathcal{S}$, instância para Supercadeia Mínima, o grafo $G_\mathcal{S}$, grafo de prefixos de $\mathcal{S}$, e $\mathcal{C}$, uma cobertura por ciclos mínima de $G_\mathcal{S}$. Sejam $C = (c_1, c_2, \ldots, c_k, c_1)$ e $D = (d_1, d_2, \ldots, d_m, d_1)$ dois ciclos em $\mathcal{C}$ e sejam c_1 e d_1 suas respectivas cadeias representativas. Então*

$$|Sobre(S_{c_1}, S_{d_1})| < |\alpha(C)| + |\alpha(D)|\,.$$

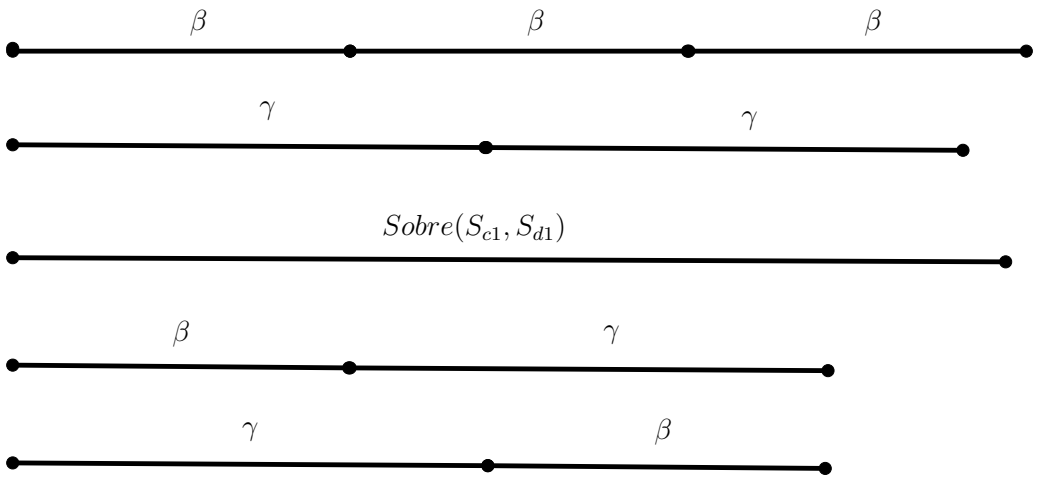
Demonstração. Vamos realizar a demonstração por contradição. Assim, suponha que $|Sobre(S_{c_1}, S_{d_1})| \geq |\alpha(C)| + |\alpha(D)|$.

Primeiro perceba que como $|S_{c_1}| > |\alpha(C)|$, então após os $|\alpha(C)|$ primeiros caracteres de S_{c_1}, há repetição desses mesmos primeiros caracteres. Isso ocorre ao longo de toda a cadeia S_{c_1}. O mesmo vale para S_{d_1} e $\alpha(D)$. Isso implica que S_{c_1} é uma subcadeia de $\alpha(C)^\infty$ e S_{d_1} é uma subcadeia de $\alpha(D)^\infty$.

Vamos definir que β é a cadeia que contém os primeiros $|\alpha(C)|$ caracteres de $Sobre(S_{c_1}, S_{d_1})$ e que γ é a cadeia que contém os primeiros $|\alpha(D)|$ caracteres também de $Sobre(S_{c_1}, S_{d_1})$. Um esquema dessas definições é dado na figura a seguir.



Como $|\beta| = |\alpha(C)|$, então independente de onde $Sobre(S_{c_1}, S_{d_1})$ comece em S_{c_1} e $\alpha(c)$, vale que S_{c_1} é subcadeia de β^∞. Logo, o próprio $Sobre(S_{c_1}, S_{d_1})$ é subcadeia de β^∞. Por razões análogas, para S_{d_1}, γ e $\alpha(D)$, também vale que $Sobre(S_{c_1}, S_{d_1})$ é subcadeia de γ^∞. A figura a seguir apresenta uma visualização desses fatos.



Perceba na imagem acima que β e γ comutam, ou seja, independente da ordem que concatenarmos β e γ obtemos o mesmo resultado. Logo concluímos que $\beta^\infty = \gamma^\infty$. Assim, qualquer cadeia de $x \in \mathbb{Z}^+$ caracteres em β é encontrada na mesma posição em γ.

Pelo Lema 2.10, sabemos que existe um ciclo de custo no máximo $|\beta| = |\alpha(C)|$ que cobre todos os vértices que estão em C e D. Isso é uma contradição, afinal assumimos que $\mathcal{C}$ é uma cobertura mínima por ciclos para a entrada $\mathcal{S}$. □

Com o Lema 2.11 já conseguimos demonstrar o fator de aproximação do Algoritmo 10. Segue o teorema.

Teorema 2.10. *Dados uma instância $\langle \mathcal{S} \rangle$ para o problema da Supercadeia Mínima, um grafo de prefixos de $G_{\mathcal{S}}$, uma cobertura por ciclos ótima $\mathcal{C} = \{C_1, \ldots, C_m\}$ para $G_{\mathcal{S}}$ e a supercadeia A devolvida ao final do Algoritmo 10,*

então o Algoritmo 10 é uma 4-aproximação para o problema da Supercadeia Mínima.

Demonstração. O caso em que a cadeia representativa de cada ciclo C_i em $\mathcal{C}$ é menor do que $\alpha(C_i)$ já foi discutido e concluímos que nesse caso

$$|A| \leq 2OPT_{SM}(\mathcal{S}) < 4OPT_{SM}(\mathcal{S})\,. \tag{2.24}$$

Agora, caso a cadeia representativa de algum C_i seja maior do que $\alpha(C_i)$, a relação acima já não é necessariamente verdadeira.

Sendo S_{r_i} a cadeia representativa de cada ciclo C_i, veja que

$$|A| = \sum_{i=1}^{|\Phi|} |\phi(C_i)| = \sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)| + \sum_{i=1}^{|\mathcal{C}|} |S_{r_i}|\,. \tag{2.25}$$

Sabemos que

$$\sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)| \leq OPT_{SM}(\mathcal{S})\,. \tag{2.26}$$

Vamos renomear as cadeias representativas dos ciclos em $\mathcal{C}$ segundo a ordem que elas aparecem da esquerda para direita em uma solução ótima para o problema. Assim, por construção

$$\begin{aligned} OPT_{SM}(\mathcal{S}) &= \sum_{i=1}^{|\mathcal{C}|-1} |Pref(S_{r_i}, S_{r_{i+1}})| + |Sobre(S_{|\mathcal{C}|}, S_{r_1})| \\ &= \sum_{i=1}^{|\mathcal{C}|} |S_{r_i}| - \sum_{i=1}^{|\mathcal{C}|-1} |Sobre(S_{r_i}, S_{r_{i+1}})|\,. \end{aligned} \tag{2.27}$$

Usando o Lema 2.11 e a desigualdade (2.27) chegamos à seguinte desigualdade

$$OPT_{SM}(\mathcal{S}) \geq \sum_{i=1}^{|\mathcal{C}|} |S_{r_i}| - 2\sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)|\,. \tag{2.28}$$

Com essa desigualdade concluímos que

$$\sum_{i=1}^{|\mathcal{C}|} |S_{r_i}| \leq OPT_{SM}(\mathcal{S}) + 2\sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)| \leq 3OPT_{SM}(\mathcal{S})\,, \tag{2.29}$$

sendo que a última desigualdade segue de (2.23).

Com isso demonstramos um limite superior para a soma de todas as cadeias representativas, o que nos leva a concluir que

$$|A| = \sum_{i=1}^{|\mathcal{C}|} |\alpha(C_i)| + \sum_{i=1}^{|\mathcal{C}|} |S_{r_i}| \leq 4OPT_{SM}(\mathcal{S})\,.$$

□

Cabe ainda a menção a existência de uma 3-aproximação da Supercadeia Mínima, que é mostrada por Vazirani [6].

2.7 Corte Multiseparador (*Multiway Cut*)

Para esse problema, relembre a Definição 1.5, de corte em um grafo. Dado um grafo G, vamos denotar um corte que separa, ou desconecta, os vértices u e v como $cut_G(u, v)$.

Definição 2.8 (Corte Isolado)**.** *Dado um grafo G e um conjunto $V' \in V(G)$, um* **corte isolado de** *V' é o corte cuja remoção desconecta V' de $V(G) \setminus V'$.*

Suponha que dado um grafo G desejemos remover todos os caminhos que conectam dois vértices desse grafo, ou seja, encontrar $cut_G(u, v)$ para todo par de vértices u, v. Uma forma simples de fazer isso é remover todas as arestas incidentes a u (ou a v), por exemplo. Mas e se quisermos remover todos os caminhos que conectam um determinado conjunto de vértices? A solução para isso é encontrar um *corte multiseparador*.

Definição 2.9 (Corte Multiseparador)**.** *Dado um grafo G e um conjunto $T = \{t_1, t_2, \ldots, t_k\} \subseteq V(G)$, um* ***corte multiseparador*** *de T é um conjunto de arestas $C \in E(G)$ cuja a remoção de G desconecta cada par em T.*

Chamamos o conjunto T de conjunto de terminais. Segue um exemplo de Corte Multiseparador.

Problema 2.13 (Corte Multiseparador)**.** *Dado um grafo G, um conjunto $T = \{t_1, t_2, \ldots, t_k\} \in V(G)$ e uma função $c : E(G) \to \mathbb{Q}^+$, buscamos um corte multiseparador C de T que possua custo mínimo, isto é, com menor valor $c(C) = \sum_{e \in C} c(e)$.*

Para uma quantidade de terminais maior ou igual a 3 o problema do Corte Multiseparador é $\mathcal{NP}$-difícil [6].

Vemos que uma instância do problema é composta por um grafo G, um conjunto de vértices terminais que pertencem a G e uma função de custo nas arestas. Já uma solução viável do problema é composta por qualquer corte

que isole cada vértice do conjunto de terminais dos outros vértices terminais. É evidente que esse é um problema de minimização.

Seja k a quantidade de terminais que desejamos isolar. Apresentaremos um algoritmo de aproximação para o Corte Multiseparador com fator de aproximação $2 - \frac{2}{k}$. Ele é formalizado no no Algoritmo 11.

Algoritmo 11: CORTE-MULTISEPARADOR(G, T, c)

Entrada: Grafo G, conjunto $T \subseteq V(G)$ e função $c : E(G) \to \mathbb{Q}^+$
Saída: Corte multiseparador de T

```
1 Seja T = {t_1, t_2, ..., t_k}
2 para i = 1 até k faça
3 |  C_i = CORTE-ISOLANTE(G, t_i, c)
4 fim
5 C_j = max_{i∈{1,2,...,k}} c(C_i)
6 C = ⋃_{i≠j} C_i
7 retorna C
```

No laço **para** do algoritmo computa-se um corte isolante para cada vértice terminal em T, sendo que cada C_i representa um corte isolado do vértice t_i. Depois de computarmos os cortes isolados de todos os vértices terminais, selecionamos aquele com maior custo e o chamamos de C_j. Devolvemos então a união de todos os C_i exceto por C_j.

Veja que claramente C é um corte multiseparador dos vértices terminais, já que isola $k-1$ vértices do resto do grafo, garantindo que mesmo o vértice t_j (cujo C_j não está contido em C) esteja isolado dos outros vértices terminais.

Computar um corte isolado de custo mínimo de apenas um vértice pode ser feito em tempo polinomial [6], assim como os processos de selecionar o corte isolado mais custoso e unir os cortes isolados em um único conjunto, de forma que o Algoritmo 11 é polinomial. Segue o teorema sobre seu fator de aproximação.

Teorema 2.11. *O Algoritmo 11 é uma $\left(2 - \frac{2}{k}\right)$-aproximação para o problema do Corte Multiseparador.*

Demonstração. Sejam $I = \langle G, T, c \rangle$ uma instância do Corte Multiseparador e A um corte multiseparador ótimo para tal instância ($c(A) = OPT_{CM}(I)$).

Veja que $G - A$ possui $|T| = k$ componentes conexas, sendo que cada componente possui um terminal em T. Dessa forma, podemos interpretar A como a união de k cortes isolantes, um para cada componente. Seja $T = \{t_1, t_2, \ldots, t_k\}$ e vamos denotar por A_i o corte isolante da componente que contém t_i e que seja derivado de A. Dessa forma,

$$A = \bigcup_{i=1}^{k} A_i \,.$$

Cada aresta que pertence a A incide em duas componentes. Logo, cada aresta em A está em dois cortes A_i, de forma que

$$\sum_{i=1}^{k} c(A_i) = 2 \cdot c(A) \,.$$

Cada corte A_i isola uma componente e também o vértice terminal t_i que está contido nessa componente. Então A_i é corte isolante de t_i. No Algoritmo 11, computamos para cada vértice terminal t_i um corte isolante C_i de custo mínimo. Assim, podemos afirmar que

$$\sum_{i=1}^{k} c(C_i) \leq \sum_{i=1}^{k} c(A_i) = 2 \cdot c(A) \,.$$

Lembre-se que C é a união de todos os cortes C_i, exceto pelo mais custoso, que é nomeado de C_j. Como C_j é o corte mais custo, então

$$\frac{1}{k} \sum_{i=1}^{k} c(C_i) \leq c(C_j) \,,$$

pois a média de um conjunto de valores sempre é menor ou igual que o elemento mais custoso do conjunto.

Podemos então concluir que

$$c(C) = \sum_{i=1}^{k} c(C_i) - c(C_j) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} c(C_i)$$
$$\leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} c(A_i) = \left(2 - \frac{2}{k}\right) c(A)\,.$$

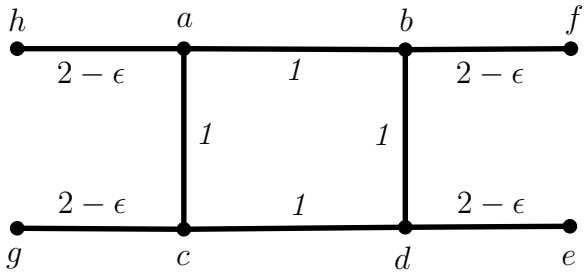
Como $c(A) = OPT_{CM}(G, T, c)$, então

$$c(C) \leq \left(2 - \frac{2}{k}\right) OPT_{CM}(G, T, c)\,.$$

□

Segue um exemplo que mostra que o fator de aproximação do algoritmo Corte-Multiseparador é justo.

Exemplo 2.9. *O grafo abaixo apresenta* 8 *vértices e o custo para cada uma de suas arestas.*



Desejamos encontrar um corte multiseparador para esse grafo, em que os vértices terminais são $T = \{e, f, g, h\}$.

Se usarmos o algoritmo Corte-Multiseparador *para resolver o problema, encontraremos os cortes isolantes mínimos para cada vértice terminal.*

O corte isolante mínimo de cada vértice terminal resume-se à aresta de custo $2 - \epsilon$ *com um extremo em tal vértice. Assim, o corte* C *devolvido pode ser* $\{ah, cg, bf\}$*, que tem custo* $6 - 3\epsilon$.

Note que a solução ótima para esse problema consiste nas 4 *arestas de custo* 1 *do grafo, ou seja,* $OPT = 4$.

Assim, quanto menor ϵ, *mais o fator de aproximação dessa instância se aproxima de* $2 - \frac{2}{k}$.

2.8 k-Corte (k-*Cut*)

O problema do k-corte, assim como o Problema do Corte Multiseparador (veja Seção 2.7), envolve o conceito de corte em um grafo (Definição 1.5), já que está relacionado à minimização do custo de um corte. Segue uma descrição formal do problema.

Problema 2.14 (k-Corte Mínimo)**.** *Dados um grafo G, uma função $c : E(G) \to \mathbb{R}^+$ e um valor k, desejamos encontrar um conjunto de arestas $E' \subseteq E(G)$ tal que o grafo $G - E'$ possua k componentes conexas e $c(E') = \sum_{e \in E'} c(e)$ seja mínimo.*

Da mesma forma que no problema do corte multiseparador (Seção 2.7), mostraremos uma $\left(2 - \frac{2}{k}\right)$-aproximação para o problema do k-corte.

Perceba que uma instância do Problema 2.14 é composta por um grafo G, uma função de custo $c : E(G) \to \mathbb{R}^+$ nas arestas e um valor k. Denotamos o custo de uma solução ótima por $OPT_{KCut}(I)$.

O algoritmo em questão usa uma árvore de Gomory-Hu para representar o grafo da instância. Antes de definirmos uma árvore Gomory-Hu temos que definir o que é um corte associado a uma aresta. Lembre-se que dado $X \subseteq V(G)$, podemos escrever o corte $\partial_G(X)$ como $\{X, V(G) \setminus X\}_G$.

Definição 2.10 (Corte associado à aresta)**.** *Sejam G um grafo e T uma árvore em que $V(T) = V(G)$ mas não necessariamente $E(T) \subseteq E(G)$. Para $e \in E(T)$, denote por S e S' os conjuntos de vértices das componentes conexas de $T - e$. O corte $\{S, S'\}_G$ é o* **corte associado a** e **em** G*.*

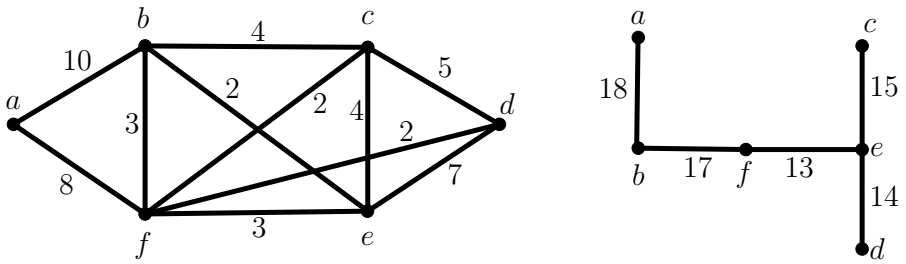
Definido o conceito de corte associado à aresta, vamos agora definir o que é uma árvore de Gomory-Hu.

Definição 2.11 (Árvore de Gomory-Hu)**.** *Sejam G um grafo e T uma árvore em que $V(T) = V(G)$ mas não necessariamente $V(T) \subseteq V(G)$. Sejam $c : E(G) \to \mathbb{R}^+$ e $c' : T(G) \to \mathbb{R}^+$. Dizemos que T é uma* **árvore de Gomory-Hu** *de G se:*

- $\forall\ u, v \in V(G)$, *a aresta de menor custo do caminho de* u *a* v *em* T *(isto é, o custo de um corte mínimo que separa* u *e* v *em* T*) é igual ao custo de um corte mínimo que separa* u *e* v *em* G*;*

- $\forall\ e \in E(T)$, $c'(e)$ *é o peso do corte associado a* e *em* G*.*

Abaixo temos um exemplo de um grafo e de sua respectiva árvore Gomory-Hu:



Por exemplo, na árvore (à direita), o custo do menor corte que separa a e d é 13 (menor aresta no caminho entre eles), que é também o custo do menor corte que separa a e d no grafo, que é o corte $\{bc, be, cf, ef, df\}$.

A construção de uma árvore Gomury-Hu é feita por um algoritmo em tempo polinomial [6].

Vamos agora enunciar um lema que será usado na demonstração do fator de aproximação do algoritmo.

Lema 2.12. *Seja* T *á árvore Gomory-Hu de* G *e seja* $F \subseteq E(G)$ *o conjunto dos cortes associados a* ℓ *arestas de* T*. O grafo* $G - F$ *tem ao menos* ℓ *componentes conexas.*

Demonstração. Remover as ℓ arestas de T deixa exatamente $\ell + 1$ componentes conexas, sendo que cada uma contém um subconjunto dos vértices de T, denotados $V_1, \ldots, V_{\ell+1} \subseteq V(T)$. Claramente, remover F de G vai desconectar cada par de subconjuntos V_i, V_j deixando, assim, ao menos $\ell + 1$ componentes em G. □

Veja que esse último lema nos garante que se retirarmos de $k-1$ cortes associados a arestas de uma árvore Gomory-Hu de G, então o grafo resultante possuirá ao menos k componentes conexas. Com essa informação já é possível compreender o funcionamento do Algoritmo 12.

Algoritmo 12: MINIMO_kCUT(G, c, k)

Entrada: Grafo G, função c de custo nas arestas, valor k
Saída: Conjunto C de arestas

```
1 T, c' = GOMORY_HU(G, c) /* gerar árvore Gomory-Hu de G */
2 C = ∅
3 para i = 1 até k − 1 faça
4 |   C_i = corte associado à i-ésima aresta de menor custo de T
5 |   C = C ∪ C_i
6 fim
7 retorna C
```

Um detalhe muito importante na hora de implementar esse algoritmo é conferir se o grafo $G - C$ possui exatamente k componentes, pois, pelo Lema 2.12, ele pode ter mais. Caso possua mais componentes, deve-se adicionar de volta algumas arestas de C até obtermos as k componentes desejadas.

Agora segue o Teorema 2.12, sobre o fator de aproximação do Algoritmo 12.

Teorema 2.12. *O Algoritmo 12 é uma aproximação com fator* $2 - \frac{2}{k}$ *para o Problema do k-Corte Mínimo.*

Demonstração. Seja $I = \langle G, c, k \rangle$ uma instância do problema e seja A um k-corte ótimo de G , ou seja, $c(A) = OPT_{KCut}(I)$.

Podemos interpretar um k-corte ótimo como a união de k cortes distintos: para cada um dos k conjuntos de vértices $V_1, \ldots, V_k$ das componentes conexas de $G - A$ é possível definir k cortes isolados que utilizam arestas de A.

Denote o corte isolado de V_i por A_i. Dessa forma, $A = A_1 \cup \cdots \cup A_k$. Sem perda de generalidade, assuma que A_k é o corte com maior custo.

Veja que cada aresta de um corte A_i possui extremidade em duas componentes conexas de $G - A$. Assim, é fácil perceber que

$$\sum_{i=1}^{k} c(A_i) = 2\, c(A)\,.$$

Seja T a árvore de Gomory-Hu gerada no Algoritmo 12. Vamos definir B como o conjunto de arestas uv de T em que $u \in V_i$ e $v \in V_j$, $j \neq i$ e c' como a função que determina o custo de cada uma dessas arestas. Note que $|B| \geq k - 1$.

Vamos agora realizar uma espécie de redução da árvore Gomory-Hu T para uma outra árvore H, onde:

- Cada conjunto V_i representa um vértice v_i em H, sendo que $V(H) = \{v_1, \ldots, v_k\}$;
- A escolha das arestas de $E(H)$ é arbitrária desde que atenda o requisito de H ser árvore.

Por ter suas arestas originadas de uma árvore Gomory-Hu de G, cada aresta em H representa um corte em G. Como $|E(H)| = k - 1$ então, temos $k - 1$ cortes associados às arestas de H.

Vamos fazer a seguinte associação entre arestas e vértices de H. Direcione as arestas de H de forma que v_k, vértice associado à componente mais pesada do corte ótimo, seja a raiz (grau de entrada zero). Agora associe a cada v_i a aresta direcionada a dele.

Assim, cada um dos $k-1$ vértices $v_1, \ldots, v_{k-1}$ de H está associado a uma única aresta de $E(H)$.

Por T ser uma árvore Gomory-Hu, cada uma de suas arestas possui um corte mínimo associado em G. Essa propriedade acaba sendo passada para H devido a maneira que H foi definida. Isso quer dizer que cada uma das $k - 1$ arestas v_iv_j de H está associada a um corte mínimo entre os conjuntos de vértices V_i e V_j em G.

Vamos denotar esses cortes mínimos que separam as componentes V_i e V_j pela aresta v_iv_j.

Como A_i é um corte isolado mínimo da componente V_i, por consequência A_i é corte que separa V_i e V_j. Logo vale que

$$c'(v_iv_j) \leq c(A_i)\,.$$

Perceba que essa última desigualdade vale para todo corte isolado A_i.

Sabemos que o conjunto C, devolvido pelo algoritmo é o conjunto das $k-1$ arestas de menor custo de T, representando $k-1$ cortes mínimos em G, dessa forma vale que

$$c(C) \leq \sum_{e \in B} c'(e)\,.$$

Já que toda aresta de B está contida no conjunto de arestas de T, temos que

$$\sum_{e \in B} c'(e) \leq \sum_{i=1}^{k-1} c(A_i),$$

pois todo corte isolante A_i é mais custoso que o corte mínimo que separa V_i (representado pelas arestas de B que têm extremos em V_i). Então,

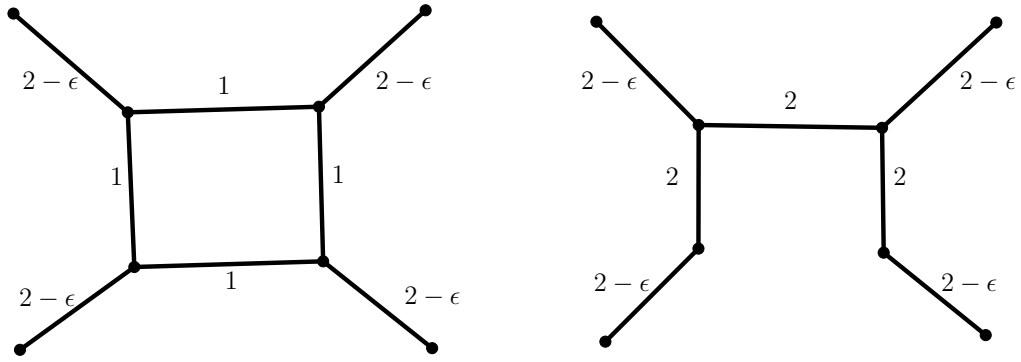
$$\sum_{i=1}^{k-1} c(A_i) \leq \left(1 - \frac{1}{k}\right) \sum_{i=1}^{k} c(A_i) = 2\left(1 - \frac{1}{k}\right) c(A)\,.$$

Sendo que a desigualdade vale porque $c(A_k)$ é sempre maior ou igual à media de custo dos cortes. □

Feita a demonstração e considerando que a função GOMORY_HU(G) possui tempo polinomial, veja que o tempo do resto algoritmo não ultrapassa $O(n^2)$, já que o único laço **para** do algoritmo realiza $k-1$ interações, sendo que $k-1 < n$, e há um custo extra para encontrar as arestas de menor custo a cada passo.

Agora vamos a um exemplo justo de execução do Algoritmo 12.

Exemplo 2.10. *Veja os dois grafos abaixo. À esquerda temos um grafo G com os custos de cada uma de suas arestas, onde ϵ representa um valor infinitesimal. À direita temos a árvore Gomory-Hu de G.*



Se nossa instância é o grafo G e $k = 4$, então o Algoritmo 12 ao escolher as $k - 1$ arestas de menor custo escolherá as três arestas de custo $2 - \epsilon$, o que leva o custo total da solução a ser $6 - 3\epsilon$.

Porém veja que o k-cut mínimo para G, com $k = 4$, é dado pelas quatro arestas centrais de custo 1 no grafo.

Atentando-se as relações do custo da solução A devolvida pelo algoritmo e da solução ótima, obtemos a seguinte relação:

$$c(A) = 6 - 3\epsilon = \left(2 - \frac{2\epsilon}{4}\right) 4 = \left(2 - \frac{2\epsilon}{k}\right) OPT_{KCut}$$

A relação acima é infinitesimalmente próxima do fator de aproximação.

2.9 k-Centro (k-*Center*)

Nessa seção falaremos do k-Centro, um problema que não pode ser aproximado por um algoritmo de tempo polinomial a não ser que $\mathcal{P} = \mathcal{NP}$ [6], por isso falaremos sobre a versão métrica do k-Centro. A formalização do k-centro métrico é dada no Problema 2.15.

Em especial, apresentaremos nessa seção dois resultados referentes ao k-centro métrico:

- Uma 2-aproximação para o problema;
- Uma 2-aproximação é a melhor aproximação possível para o problema se $\mathcal{P} \neq \mathcal{NP}$.

Vamos agora definir uma notação que será usada na definição do problema do k-centro métrico. Dados um grafo completo G, uma função de custo nas arestas de G, um conjunto $S \subset V(G)$ e um vértice $V \in V(G)$, vamos denotar por $conecta(v, S)$ o custo da aresta menos custosa que conecta algum vértice de S a v.

Problema 2.15 (k-Centro Métrico)**.** *Sejam G um grafo completo, c: $E(G) \to \mathbb{Q}^+$ uma função de custo nas arestas que respeite a desigualdade triangular e $k \in \mathbb{Z}^+$. Desejamos encontrar um conjunto $S \subseteq V(G)$, onde $|S| = k$, de forma que $\max_{v \in V(G) \setminus S}\{conecta(v, S)\}$ seja mínimo.*

A partir daqui vamos nos referir à versão métrica do k-centro apenas como k-Centro. Veja que o Problema 2.15 é um problema de minimização, sendo que uma instância desse problema é composta por um grafo completo, uma função de custo nas arestas que respeite a desigualdade triangular e um inteiro positivo.

A estratégia que apresentaremos para resolver o k-Centro envolve uma técnica chamada ***poda paramétrica*** [6].

Antes de falarmos sobre tal técnica, perceba que se soubermos o custo de uma solução ótima, então podemos eliminar partes da entrada que nunca

seriam usadas em uma solução ótima. O problema dessa ideia é justamente encontrar o custo da solução ótima.

Na poda paramétrica, ao invés de usarmos o custo da solução ótima, usamos um parâmetro t, cuja seleção varia de problema para problema. Segue uma descrição dos passos da técnica de poda paramétrica:

1. Um parâmetro t é escolhido;

2. A instância do problema é podada, sendo removidas todas as partes que não serão usadas em soluções com custo $\leq t$;

3. Usamos a instância podada para computar um limitante inferior no custo da solução ótima;

4. Encontramos uma solução com custo até α vezes maior que o limitante inferior encontrado, para algum α coerente com nossos objetivos.

Para aplicarmos a poda paramétrica no problema do k-Centro vamos primeiro ordenar de maneira não decrescente as arestas do grafo G segundo seu custo. Também vamos renomear as arestas de G usando sua posição na ordenação realizada. Então, $E(G) = \{e_1, \ldots, e_m\}$, onde para todo $1 \leq i < m$ vale que $c(e_i) \leq c(e_{i+1})$ e $m = |E(G)|$.

Vamos denotar por G_i o grafo tal que $V(G_i) = V(G)$ e $E(G_i) = \{e_1, \ldots, e_i\}$, sendo que $i \leq m$.

Para prosseguir, vamos definir o que é um conjunto dominante.

Definição 2.12 (Conjunto Dominante)**.** *Dado um grafo G, dizemos que $S \subseteq V(G)$ é* **conjunto dominante** *em G se $\forall\ v \in V(G) \setminus S$, $\exists vw \in E(G)$ tal que $w \in S$.*

Vamos denotar por $\gamma(G)$ a cardinalidade de um conjunto dominante de tamanho mínimo em um grafo G. Encontrar $\gamma(G)$ para um grafo G qualquer é um problema $\mathcal{NP}$-difícil [6].

Veja que resolver um k-centro é equivalente a encontrar o menor índice i tal que G_i possua ao menos um conjunto dominante de tamanho k. Isso é verdade, pois quando encontramos tal G_i, já sabemos que ele tem um conjunto dominante de tamanho k, logo um k-centro, e sabemos que o custo dessas arestas que compõem esse k-centro é o mínimo possível devido à forma que G_i é construído.

Dessa forma, se i^* é esse menor índice, então $c(e_{i^*})$ é o custo de um k-centro ótimo, já que com certeza usaremos essa aresta no k-centro construído, pois, caso contrário, significaria que poderíamos ter construído um k-centro em algum G_i tal que $i \leq i^*$.

Veja que dado um k-centro S sobre um grafo G, podemos particionar os vértices de G em conjuntos S e $V(G) \setminus S$, onde $|S| = k$. Veja que as arestas internas[4] de S ou $V(G) \setminus S$ não são consideradas no custo do problema, pois estamos interessados nas arestas que têm um extremo em S e o outro em $V(G) \setminus S$. Então, considerando essa relação entre os vértices de S e $V(G) \setminus S$, podemos dizer que cada vértice de S é o centro de uma ***estrela*** e todas essas estrelas juntas são uma ***floresta geradora do grafo*** G.

Para entender melhor o conceito de estrelas discutido no último parágrafo, observe o k-centro sobre um grafo G na Figura 2.2. As arestas pontilhadas nessa figura representam as arestas consideradas internas nos conjuntos S e $V(G) \setminus S$, ou seja, aquelas que não participam do k-centro considerado.

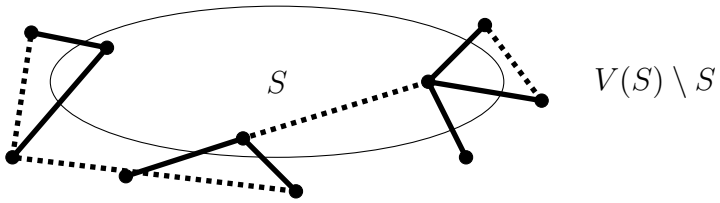


Figura 2.2: Exemplo de representação do conceito de estrelas em um k-centro usando um grafo G particionado em S e $V(G) \setminus S$.

[4]Interna no sentido de conectar dois vértices de S ou dois vértices de $V(G) \setminus S$.

Vamos chamar o conjunto de estrelas geradoras de um grafo G_i de ***estrelas geradoras de*** G_i ***com relação a*** S. Assim, encontrar o índice i tal que G_i tem um conjunto dominante de tamanho no máximo k é equivalente a encontrar G_i que contenha k estrelas geradoras [6]. Falamos no máximo e não igual à k, pois caso tenhamos um conjunto dominante com menos do k elementos podemos adicionar os outros vértices ao conjunto, já que isso não vai contra a definição de conjunto dominante.

Vamos agora definir o que são o quadrado de um grafo, um conjunto independente de um grafo e uma clique em um grafo. Esses conceitos serão usados no algoritmo que apresentaremos e nas demonstração do teorema que refere-se ao fator de aproximação do algoritmo.

Definição 2.13 (Quadrado de um Grafo)**.** *Dado um grafo H, definimos como o* **quadrado** *de H o grafo H^2, onde $V(H^2) = V(H)$, mas dados $u, v \in V(G)$, para H^2 existe uma aresta uv sempre que H possuir um caminho de tamanho no máximo dois entre u e v, onde $u \neq v$.*

Definição 2.14 (Conjunto Independente)**.** *Dado um grafo G e um conjunto $T \subseteq V(G)$, falamos que T é* **conjunto independente** *de G se nenhum par de vértices $v, w \in T$ é adjacente entre si.*

Definição 2.15 (Clique)**.** *Dado um grafo G e um conjunto $T \subseteq V(G)$, falamos que T é* **clique** *de G se todo par de vértices $v, w \in T$ é adjacente entre si.*

Segue um resultado que será usado para demonstrar um limitante inferior para $OPT_{KC}(G, c, k)$ e também serve de base para uma tomada de decisão no Algoritmo **??**.

Lema 2.13. *Dados um grafo H e um conjunto independente I em H^2, então $|I| \leq \gamma(H)$.*

Demonstração. Seja D um conjunto dominante mínimo em H. Então H contém um conjunto de estrelas geradoras de tamanho $|D|$. Perceba que

cada estrela em H forma uma clique em H^2, assim, H^2 possui $|D|$ cliques geradoras. Agora perceba que nesse caso qualquer conjunto independente em H^2 pode conter no máximo um vértice por clique, fazendo o lema valer. □

Finalmente, apresentamos o Algoritmo 13 que possui fator 2 de aproximação para o problema do k-Centro, conforme o Teorema 2.13 prova.

Algoritmo 13: KCentro(G, c, k)

Entrada: Grafo G, função de custo nas arestas c e inteiro positivo k
Saída: Conjunto M de k vértices de G

```
para i = 1 até |E(G)| faça
    Seja M_i um conjunto independente maximal em G_i^2
fim
Seja h o menor i tal que |M_i| ≤ k
retorna M_h
```

Uma informação relevante sobre o funcionamento desse algoritmo é que encontrar um conjunto independente maximal é um processo que leva tempo polinomial [6].

O lema a seguir relaciona o custo de uma solução ótima com o custo da aresta de maior custo em G_h, onde h é como definido no algoritmo. Mas antes, lembre-se que um grafo G_i^2 possui arestas que o grafo G_i não possuiria, então $\max_{v \in V(G_i^2) \setminus S}\{conecta(v, S)\} = c(e_h)$ pode ser diferente do custo da solução ótima do problema.

Lema 2.14. *Seja h como definido no Algoritmo 13, temos que $c(e_h) \leq OPT_{KC}(G, c, k)$.*

Demonstração. Para cada $i < h$ sabemos que $|M_i| > k$. Pelo Lema 2.13 sabemos que para esses valores de i temos que $\gamma(G_i) \geq |M_i| > k$. Seja i^* o índice que possibilitaria a construção de uma solução ótima para o k-centro. Como h é o menor índice que possibilitaria uma solução viável, então $i^* \geq h$, assim, $c(e_{i*}) \geq c(e_h)$. □

Teorema 2.13. *O Algoritmo 13 é uma 2-aproximação para o problema do k-Centro Métrico.*

Demonstração. Perceba que um conjunto independente maximal L em um grafo G é também um conjunto dominante em G, logo os conjuntos M_i do algoritmo são conjuntos dominantes em G_i^2.

Veja que existe um conjunto de k estrelas geradoras centradas nos vértices de M_h que cobrem todos os vértices de G.

Pela desigualdade triangular, cada aresta usada na construção de G_h^2, usando como base as estrelas geradoras, possui custo de no máximo $2c(e_h)$, assim,

$$\max_{v \in V(G) \setminus M_h} \{conecta(v, M_h)\} \leq 2c(e_h) \leq 2OPT_{KC}(G, c, k) \ .$$

□

Outro resultado relevante sobre o problema do k-Centro Métrico é de que se $\mathcal{P} \neq \mathcal{NP}$ a melhor aproximação para esse problema possui fator 2. Esse resultado é enunciado no Teorema 2.14. A prova usada nesse teorema é semelhante à usada na demonstração da inaproximabilidade do TSP na Seção 2.4, pois reduz um problema conhecido $\mathcal{NP}$-completo e o resolve a partir da suposta aproximação do k-center. O problema que será reduzido é o problema do Conjunto Dominante, um problema de decisão, cuja formalização é dada a seguir.

Problema 2.16 (Conjunto Dominante)**.** *Dados um grafo H e um valor $k \in \mathbb{Z}^+$, desejamos saber se existe* **conjunto dominante** *em H de tamanho no máximo k.*

Teorema 2.14. *Dado $\epsilon > 0$, não existe algoritmo de aproximação de tempo polinomial para o problema do k-centro métrico com fator $2 - \epsilon$, a não ser que $\mathcal{P} = \mathcal{NP}$.*

Demonstração. Seja $\langle G, k\rangle$ uma instância do problema do menor conjunto dominante, onde G é um grafo e $k \in \mathbb{Z}^+$.

Vamos construir o grafo completo G' onde $V(G') = V(G)$ e uma função c de custo nas arestas de G' da seguinte forma:

$$c(uv) = \begin{cases} 1, & \text{se } uv \in E(G) \\ 2, & \text{se } uv \notin E(G) \ . \end{cases}$$

É evidente que G' satisfaz a desigualdade triangular. Assim, construímos $\langle G', c, k\rangle$, uma instância do k-centro métrico.

Assuma que A é um algoritmo de aproximação para o problema do k-Centro Métrico com fator de aproximação $2 - \epsilon$, onde $\epsilon > 0$. Assim sendo, ao aplicarmos nosso algoritmo A no grafo G' perceba que só podemos dividir a resposta obtida em dois casos:

- Se A devolve um k-centro de custo 1, então podemos construir um conjunto dominante de tamanho no máximo k em G e, assim, $\gamma(G) \leq k$;
- Se A devolve um k-centro de custo 2, então não podemos construir um conjunto dominante de tamanho no máximo k em G e, assim, $\gamma(G) > k$.

Veja que essas conclusões tiradas do custo do k-centro devolvido são verdadeiras, pois um k-centro é um conjunto dominante e se não podemos encontrar um no grafo G, então o tamanho do menor conjunto dominante em G é maior que k. Assim, resolvemos um problema $\mathcal{NP}$-difícil em tempo polinomial, logo $\mathcal{P} = \mathcal{NP}$. □

Capítulo 3

Esquemas de Aproximação em Tempo Polinomial

Nesse capítulo falaremos sobre esquemas de aproximação. A ideia associada a esse tipo de algoritmo é que executá-lo por mais tempo pode levar a uma resposta mais próxima da solução ótima. Assim, a ideia do esquema de aproximação é basicamente uma troca onde podemos encontrar uma resposta tão boa quanto quisermos dependendo do tempo de execução. As principais referências nessa seção serão os livros de Vazirani [6] e Williamson e Shmoys [7].

Segue uma definição que formaliza a ideia de esquemas de aproximação.

Definição 3.1 (Esquemas de Aproximação em Tempo Polinomial)**.** *Dados um problema computacional e uma instância I para esse problema, dizemos que tal problema admite um* **esquema de aproximação em tempo polinomial** *se para qualquer valor do parâmetro $\epsilon > 0$, existe um algoritmo que resolve o problema e seja uma*

- $(1 + \epsilon)$*-aproximação polinomial com relação ao tamanho de I, se o problema for de minimização; ou*
- $(1 - \epsilon)$*-aproximação polinomial com relação ao tamanho de I, se o*

problema for de maximização.

Usualmente nos referimos a um esquema de aproximação em tempo polinomial por PTAS (*Polynomial Time Approximation Scheme*). Veja que a definição acima não determina qual a relação do tempo consumido pelo PTAS com o valor de ϵ. Isso significa que podemos ter um PTAS que consuma tempo polinomial em relação ao tamanho da instância, mas que ao mesmo tempo não seja polinomial em relação ao valor de ϵ. Para um esquema de aproximação que também tenha tempo polinomial em relação a ϵ necessitamos de uma definição um pouco mais restrita.

Definição 3.2 (Esquema de Aproximação em Tempo Completamente Polinomial)**.** *Dado um problema computacional e uma instância I para tal problema, dizemos que esse problema admite um* **esquema de aproximação em tempo completamente polinomial***, se para qualquer valor do parâmetro $\epsilon > 0$, existe um algoritmo que resolve o problema e seja uma*

- $(1+\epsilon)$*-aproximação polinomial com relação ao tamanho de I e ao valor $1/\epsilon$, se o problema for de minimização; ou*
- $(1-\epsilon)$*-aproximação polinomial com relação ao tamanho de I e ao valor $1/\epsilon$, se o problema for de maximização.*

Um Esquema de Aproximação em Tempo Completamente Polinomial normalmente é referido como um FPTAS (*Fully Polynomial Time Approximation Scheme*). Caso $\mathcal{P} \neq \mathcal{NP}$, um FPTAS é o que podemos conseguir de melhor para problemas $\mathcal{NP}$-difíceis [6].

Veja que um esquema de aproximação pode ser considerado uma família de algoritmos que resolvem um determinado problema, porém, para cada ϵ, mesmo que a instância seja a mesma, podemos ter diferentes tempos de execução. No início da seção quando falamos sobre sacrificar tempo em relação a otimalidade nos referíamos a essa característica dos esquemas de aproximação, onde a escolha de um ϵ que nos deixe mais perto da solução ótima acaba levando a um consumo de tempo maior.

3.1 Mochila (*Knapsack*)

Antes de falarmos de maneira mais específica do problema da Mochila vamos falar um pouco sobre ***Programação Dinâmica***. Trata-se de uma importante técnica de construção de algoritmos que baseia-se em construir uma solução para uma instância de um problema A usando de soluções de instâncias menores do mesmo problema A.

A característica marcante da Programação Dinâmica é o uso de tabelas, ou de outras estruturas de dados, para salvar os resultados dos subproblemas e depois usá-los, impedindo o recálculo dos mesmo subproblemas. Podemos, inclusive, usar Programação Dinâmica para construir algoritmos de aproximação.

Agora, voltemos nossas atenções ao problema da mochila. Suponha que tenhamos um conjunto de objetos de valores e tamanhos distintos e uma mochila para carregar tais objetos, porém, sem espaço suficiente para carregar todos os objetos. Nosso objetivo aqui é maximizar o valor que iremos carregar, mas sempre respeitando o espaço limitado da mochila. Segue uma formalização do problema.

Problema 3.1 (Mochila)**.** *Dados um conjunto* $Obj = \{1, 2, \ldots, n\}$*, um valor* $c \in \mathbb{Z}^+$ *e as funções* $w : Obj \to \mathbb{Z}^+$ *e* $v : Obj \to \mathbb{Z}^+$*, desejamos encontrar um conjunto* $B \subseteq Obj$ *de forma que* $w(B) \leq c$ *e que* $v(B)$ *seja máximo.*

Uma instância da Mochila é dada pelo conjunto Obj de objetos a serem escolhidos, a capacidade c da mochila, além das funções w, que dá o peso dos objetos, e v, que dá o valor dos objetos. Também veja que o problema da Mochila é de maximização.

Antes de prosseguirmos vamos apresentar duas definições importantes para apresentarmos o primeiro algoritmo relacionado ao Problema 3.1. Em especial, tais definições serão importantes para notarmos as diferenças com relação aos outros algoritmos já apresentados.

Definição 3.3 (Algoritmo Pseudo-polinomial)**.** *Dados um problema X de otimização combinatória, uma instância I para tal problema e um algoritmo A que resolve esse problema, se o algoritmo A for polinomial em relação a representação unária de algum valor numérico que pertença a I, dizemos que o algoritmo A é um* **algoritmo pseudo-polinomial**.

Definição 3.4 (Fracamente $\mathcal{NP}$-Difícil)**.** *Dizemos que um problema X é* ***fracamente $\mathcal{NP}$-difícil*** *se $X \in \mathcal{NP}$-difícil, mas existe algoritmo exato e pseudo-polinomial para X.*

A razão de apresentarmos essas duas definições é porque o problema da Mochila é fracamente $\mathcal{NP}$-difícil, ou seja, possui um algoritmo exato de tempo pseudo-polinomial.

Ao longo dessa seção, considere que $I = \langle Obj, c, w, v \rangle$ é uma instância da Mochila com mesma notação daquela na definição do problema da Mochila, além disso $|Obj| = n$.

Antes de apresentarmos a ideia por trás do algoritmo exato para o problema, vamos apresentar a noção de ***par dominante***. Dizemos que um par (a, b) domina um par (c, d) se $a \leq c$ e $b \geq d$. Nesse algoritmo, um vetor A com n posições é indexado por i e será mantido de forma que cada $A[i]$ seja uma lista de pares ordenados (W, V), sendo que cada um desses pares representa, respectivamente, o peso e o valor de um conjunto de objetos $B_i \subseteq \{1, \ldots, i\}$.

A ideia do algoritmo é ir percorrendo cada elemento de A, de forma que quando o algoritmo estiver em $A[i]$ ele vai construir o conjunto $B_i \subseteq \{1, \ldots, i\}$ que possua o maior valor possível. Dessa forma, a intuição do algoritmo é que os pares dominantes, ou seja, com maior valor, tenham prioridade e sejam mantidos até o final da execução. O algoritmo que acabou de ser descrito é o Algoritmo 14.

O conceito de programação dinâmica surge nesse algoritmo porque no momento em que ele está em $A[i]$, construindo o conjunto B_i, o mesmo processo já foi feito de $A[0]$ até $A[i-1]$, que são instâncias menores do mesmo problema.

Algoritmo 14: MOCHILA(n, v, w, c)

Entrada: Quantidade n de objetos disponíveis, função valor v, função peso w e tamanho da mochila c

Saída: Valor da solução ótima

```
1  Seja A é um vetor com n posições indexado por i ∈ {1, ..., n}
2  A[1] = {(0, 0), (w(1), v(1))}
3  para i = 2 até n faça
4  |  A[i] = A[i − 1]
5  |  para todo (W, V) ∈ A[i − 1] faça
6  |  |  se W + w(i) ≤ c então
7  |  |  |  Adicione (W + w(i), V + v(i)) à lista A[i]
8  |  |  fim
9  |  fim
10 |  Ordene os pares de A[i], de forma não decrescente, considerando o
   |   peso do par.
11 |  Vamos denotar por (W_k, V_k) o k-ésimo par da lista A[i] ordenada.
12 |  (W_atual, V_atual) = (W_1, V_1)
13 |  para j = 2 até |A[i]| faça
14 |  |  se V_atual ≥ V_j então
15 |  |  |  Remova o par (W_j, V_j) de A[i]
16 |  |  fim
17 |  |  senão
18 |  |  |  (W_atual, V_atual) = (W_j, V_j)
19 |  |  fim
20 |  fim
21 fim
22 Seja (W_Max, V_Max) o par (W, V) em A[n] com maior valor de V
23 retorna V_Max
```

Não é difícil perceber que o Algoritmo 14 consome tempo $O(n \cdot \min(c, V))$, onde c é o tamanho da mochila e $V = \sum_{i \leq n} v(i)$. Basta ver que o laço **para** é iterado n vezes e os processos de construção e remoção dos pares ordenados de uma lista levam cada um até $\min(c, V)$ iterações.

Como as representações de c e V são feitas em $\lg(V)$ ou $\lg(c)$ bits, o tempo do algoritmo acaba sendo pseudo polinomial, já que ele é exponencial na quantidade de bits necessária para representar valores de entrada.

Por mais que seja uma engenhosa forma de obter o resultado do problema da mochila, para valores de V ou c muito altos o algoritmo pode ser extremamente ineficiente.

Apresentaremos agora um FPTAS para o problema da mochila que usa programação dinâmica. A principal motivação para o uso desse algoritmo é termos um algoritmo de tempo polinomial, mas sem grande perda de otimalidade.

O Algoritmo 15 adapta uma instância $\langle n, v, w, c \rangle$ do problema da mochila para uma instância que seja dependente de um valor ϵ e, após isso, ele usa o Algoritmo 14 para devolver uma solução viável da instância original.

Algoritmo 15: MochilaFPTAS(n, v, w, c, ϵ)

Entrada: Quantidade n de objetos disponíveis, função valor v, função peso w, tamanho da mochila c e parâmetro ϵ
Saída: Valor de uma solução viável de $\langle n, v, w, c \rangle$

```
1 Max = max_{i≤n}{v(i)}
2 μ = ε · Max/n
3 para i = 1 até n faça
4 |   v'(i) = ⌊v(i)/μ⌋
5 fim
6 S =Mochila(n, v', w, c)
7 retorna S
```

Dado um ϵ, estabelecemos um μ que será usado como unidade de tamanho. Definimos uma nova função v' de tamanhos, com domínio em Obj, de

forma que para cada $i \in Obj$ temos que $v'(i) = \lfloor v(i)/\mu \rfloor$. O que o Algoritmo 15 faz é executar o MOCHILA sobre a instância $\langle n, v', w, c \rangle$.

Vamos definir $V' = \sum_{i \leq n} v'(i)$, ou seja, o valor análogo ao V do Algoritmo 14, porém, usado no FPTAS. Assim, o tempo de execução de MOCHILA no Algoritmo 15 é $O(n \cdot \min(c, V'))$.

Veja que $V' = \sum_{i \leq n} v'(i) = \sum_{i \leq n} \lfloor (v(i) \cdot n)/(\epsilon \cdot Max) \rfloor$, mas como $v(i)/Max \leq 1$, esse somatório é $O(n^2/\epsilon)$. Isso implica que o tempo de execução do Algoritmo 14 com a instância adaptada é $O(n^3/\epsilon)$, pois com certeza $\min(c, V')$ é $O(n^2/\epsilon)$. Dessa forma, o tempo do Algoritmo 15 é limitado superiormente por um polinômio em $(1/\epsilon)$ e em n.

Agora vamos enunciar e demonstrar o teorema que afirma que o Algoritmo MOCHILAFPTAS apresentado é de fato um FPTAS para o problema da mochila, ou seja, demonstraremos que a solução devolvida é no mínimo $(1-\epsilon)$ vezes o valor da solução ótima.

Teorema 3.1. *O Algoritmo* MOCHILAFPTAS *é um esquema completo de aproximação polinomial para o problema da Mochila.*

Demonstração. Seja $I = \langle n, v, w, c, \epsilon \rangle$ a instância recebida pelo algoritmo, sendo que $Obj = \{1, \ldots, n\}$.

Sejam S o conjunto devolvido ao final do algoritmo e O um conjunto de objetos de uma solução ótima.

Pela definição de v', sabemos que $\forall i \in Obj$ é válido que

$$\mu \cdot v'(i) \leq v(i) \leq \mu \cdot (v'(i) + 1) \Rightarrow v(i) - \mu \leq \mu \cdot v'(i)\,. \tag{3.1}$$

Veja que S é uma solução ótima para a instância I. Isso em conjunto com a relação 3.1 nos leva à seguinte relação

$$\sum_{i \in S} v(i) \geq \mu \sum_{i \in S} v'(i) \geq \mu \sum_{i \in O} v'(i) \geq \sum_{i \in O} v(i) - |O| \cdot \mu. \tag{3.2}$$

Como $|O| \leq n$, então

$$\sum_{i \in S} v(i) \geq \sum_{i \in O} v(i) - n \cdot \mu = \sum_{i \in O} v(i) - \epsilon \cdot Max. \tag{3.3}$$

Lembrando que $v(O) = OPT_{Moc}(I)$ e que $Max \leq OPT_{Moc}(I)$, concluímos que

$$\sum_{i \in S} w(i) \geq OPT_{Moc}(I) - \epsilon \cdot OPT_{Moc}(I) = (1 - \epsilon) OPT_{Moc}(I)\,. \tag{3.4}$$

Isso, juntamente com a análise de tempo já realizada, nos leva a concluir que o Algoritmo 15 é um FPTAS. □

Agora que vimos um FPTAS para o problema da mochila, perceba que a solução devolvida pelos algoritmos apresentados é o valor de um conjunto de objetos. Porém, da forma que o Problema 3.1 foi enunciado, é interessante que também obtenhamos o conjunto de objetos que compõem a solução com o custo devolvido pelo algoritmo.

Para a estratégia adotada no Algoritmo 14, podemos conseguir o conjunto de objetos da seguinte forma: considerando que tenhamos acesso ao vetor A desse algoritmo, o que faremos é realizar o caminho inverso e encontrar os objetos que compõem a solução devolvida pelo algoritmo. Para isso vamos considerar que o Algoritmo 15 devolve o par com o peso e o valor de uma solução e não somente o valor. Essa adaptação na prática não altera nada do que foi dito até aqui sobre esse algoritmo. O Algoritmo 16 demonstra com mais detalhes os passos da construção da solução.

Algoritmo 16: MochilaConstroi$(n, v, w, c, A, (W, V))$

Entrada: Quantidade n de objetos disponíveis, função valor v, função peso w, vetor de listas A e par (W, V) devolvidos pelo Algoritmo 14

Saída: Conjunto de objetos Sol com $v(Sol) = V$

```
1  (W_atual, V_atual) = (W, V)
2  para i = n até 2 faça
3  |  se (W − w(i), V − v(i)) ∈ A[i − 1] então
4  |  |  Sol = Sol ∪ {i}
5  |  |  (W_atual, V_atual) = (W − w(i), V − v(i))
6  |  fim
7  fim
8  se (W_atual, V_atual) ≠ (0, 0) então
9  |  Sol = Sol ∪ {1}
10 fim
11 retorna Sol
```

3.2 Empacotamento (*Bin Packing*)

Considere o seguinte problema.

Problema 3.2 (Empacotamento)**.** *Dados um conjunto de objetos* $Obj = \{1, 2, \ldots, n\}$ *e uma função tamanho* $t : Obj \to (0, 1]$*, desejamos empacotar todos objetos de* Obj *em pacotes de tamanho* 1 *de forma a minimizar a quantidade de pacotes usados e respeitando o tamanho dos pacotes.*

O problema do Empacotamento possui diversas aplicações em problemas logísticos [6]. Veja que uma instância do problema é composta pelo número n de objetos e a função t de tamanhos. Uma solução viável desse problema é um empacotamento dos objetos em pacotes de tamanho 1. Trata-se de um problema de minimização sobre a quantidade de pacotes usados, logo $OPT_{Emp}(n, t)$ refere-se a uma quantidade de pacotes.

Vamos prosseguir nossa análise demonstrando que existe um limite para o fator de aproximação que podemos encontrar para um algoritmo que trata desse problema. Tal resultado é enunciado no Teorema 3.2. Mas antes de prosseguirmos para esse teorema, vamos apresentar um problema de decisão relacionado ao Empacotamento.

Problema 3.3 (Partição)**.** *Dados* k *inteiros positivos* $b_1, b_2, \ldots, b_k$ *e* $B = \sum_{i=1}^{k} b_i$*. Desejamos saber se podemos particionar o conjunto* $\{1, \ldots, k\}$ *em dois conjuntos* S *e* T *de forma que* $\sum_{i \in S} b_i = \sum_{i \in T} b_i$*.*

O problema da Partição é $\mathcal{NP}$-completo [6]. Usamos uma redução da Partição para o Empacotamento para demonstrar um limitante no fator de aproximação possível para o Empacotamento no Teorema 3.2. A redução entre esses problemas é dada da seguinte forma:

- Dada uma instância $\langle b_1, \ldots, b_k, B \rangle$ para a Partição, definimos uma função t, onde $t(i) = 2b_i/B$, e assim obtemos uma instância $\langle k, t \rangle$ para o Empacotamento;

- Ao dividirmos os objetos de $\langle k, t\rangle$ em pacotes, se obtivermos dois pacotes na resposta do Empacotamento devolvemos `sim` para Partição, caso contrário devolvemos `não` para Partição;

- Essas conclusões são possíveis pois, embora cada pacote comporte até o valor 1, devido à função t definida, cada pacote comporta objetos cuja a soma do valor total é $B/2$. Então se temos exatamente dois pacotes, podemos concluir que cada um deles possui valor de $B/2$, afinal caso contrário a soma de todos os valores não resultaria em B, o que seria um absurdo.

Teorema 3.2. *Para todo $\epsilon \geq 0$, só existe algoritmo de aproximação com fator $3/2 - \epsilon$ para o problema do Empacotamento se $\mathcal{P} = \mathcal{NP}$.*

Demonstração. Suponha que o algoritmo A seja uma aproximação para o Empacotamento com fator $3/2 - \epsilon$.

Perceba que se reduzirmos o problema da Partição para o Empacotamento, podemos usar o algoritmo A para encontrar uma solução viável do problema da Partição. A solução devolvida teria fator menor que $3/2$ de distância da quantidade ótima de pacotes.

Caso a solução devolvida possua apenas dois pacotes, então devido ao fator de aproximação conseguiríamos decidir um problema $\mathcal{NP}$-completo em tempo polinomial.

Assim, o algoritmo A só existe se $\mathcal{P} = \mathcal{NP}$. □

Não é difícil obter um algoritmo de aproximação com fator 2 para o problema do empacotamento. Esse algoritmo é conhecido como *First-Fit* (Primeiro a encaixar). Ele é guloso e funciona baseado na seguinte decisão, sendo i um elemento a ser colocado em um pacote e dados os pacotes $P_1, \ldots, P_k$ existentes até o momento:

1. Tente empacotar o elemento i em cada pacote P_j até encontrar um que comporte i;

2. Caso nenhum pacote P_j comporte i crie um novo pacote P_{k+1} e coloque i nele.

Como esse capítulo tem um enfoque maior em esquemas de aproximação não vamos apresentar o algoritmo, mas devido à sua simplicidade vamos demonstrar seu fator de aproximação. Segue o teorema que enuncia o mesmo.

Teorema 3.3. *O algoritmo* First-Fit *é uma* 2*-aproximação para o problema do Empacotamento.*

Demonstração. Seja $\langle n, t\rangle$ uma instância do Empacotamento.

Veja que a soma dos pesos de todos os n objetos é um limite inferior para o valor ótimo do problema. Assim,

$$\sum_{i=1}^{n} t(i) \leq OPT_{Emp}(n, t)\,.$$

Veja que se a solução devolvida pelo First-Fit possui m pacotes, então devido ao critério de criação de novos pacotes, podemos afirmar que temos ao menos $m-1$ pacotes que estão com mais de metade de sua capacidade.

Para entender essa última afirmação veja que quando vamos criar um novo pacote sempre sabemos que o tamanho do objeto a ser alocado é muito grande para os pacotes existentes, dessa forma:

- Se o objeto a ser alocado possui tamanho maior $1/2$, então com certeza temos no máximo um pacote com tamanho menor que $1/2$, pois caso contrário o algoritmo teria alocado em apenas um pacote os objetos desses pacotes com tamanho menor que $1/2$;

- Se o objeto a ser alocado possui tamanho menor que $1/2$, então todos os pacotes já existentes possuem tamanho maior que $1/2$.

Assim,

$$\sum_{i=1}^{n} t(i) > \frac{m-1}{2}\,.$$

Unindo as duas desigualdades anteriores obtemos que

$$\frac{m-1}{2} < OPT_{Emp}(n,t) \Rightarrow m < 2OPT_{Emp}(n,t) + 1\,.$$

Como $OPT_{Emp}(n,t)$ é um número inteiro, temos

$$m \leq 2OPT_{Emp}(n,t)\,.$$

□

Agora começaremos a falar sobre o esquema de aproximação que usaremos. Para isso começaremos apresentado a definição de Esquema de Aproximação Assintótico em Tempo Polinomial (APTAS).

Definição 3.5 (Esquema de Aproximação Assintótico em Tempo Polinomial)**.** *Um* **Esquema de Aproximação Assintótico em Tempo Polinomial** *é uma família de algoritmos A_ϵ em conjunto com uma constante c. Nessa família, para cada valor $\epsilon > 0$ existe um algoritmo A_ϵ que devolve uma solução de valor no máximo $(1+\epsilon)OPT(I) + c$ para problemas de minimização, sendo que $OPT(I)$ é o valor da solução ótima para o problema tratado sobre a instância I e c é constante.*

A seguir vamos apresentar dois lemas que serão usados para demonstrar o fator de aproximação do APTAS que apresentaremos. O primeiro lema garante que existe um algoritmo polinomial que resolve o Empacotamento se a instância possui algumas restrições. Já o segundo garante que existe um algoritmo de aproximação com fator $(1+\epsilon)$ para o problema do empacotamento com algumas restrições.

Lema 3.1. *Dados $\epsilon > 0$ e $k \in \mathbb{Z}^+$ e dada uma instância $I = \langle n, t\rangle$ para o problema do Empacotamento onde todos os objetos têm tamanho maior ou igual a ϵ e que existem k valores de tamanhos diferentes. Existe um algoritmo de tempo polinomial que resolve I de maneira ótima.*

Demonstração. Veja que como só temos objetos com tamanho maior que ϵ, então com certeza o número de objetos em cada pacote será menor do que $\lfloor 1/\epsilon \rfloor = m$, uma constante.

Como para compor cada pacote usamos até m objetos sendo que temos k tamanhos de objeto distintos, então existem $\binom{m+k}{m} = r$ composições de pacote possíveis, outra constante.

Como temos n objetos, então qualquer solução viável terá até n pacotes. Então, podemos concluir que existem menos do que $\binom{n+r}{r} = p$ soluções viáveis, valor que é polinomial em n. Logo, listar todas as soluções viáveis e escolher uma que tenha menor quantidade de pacotes garante encontrar uma solução ótima em tempo polinomial, o que nos permite afirmar que existe algoritmo polinomial em n que resolve o empacotamento para I de maneira ótima. □

Lema 3.2. *Dado $\epsilon > 0$ e dada uma instância $I = \langle n, t \rangle$ do Empacotamento onde todos os objetos possuam tamanho maior ou igual a ϵ. Existe algoritmo de aproximação que devolve uma solução para o empacotamento com fator de aproximação $(1 + \epsilon)$ para I.*

Demonstração. Primeiro, vamos ordenar os n objetos segundo seu tamanho de forma crescente. Após isso vamos particionar os objetos ordenados em $\lceil 1/\epsilon^2 \rceil = k$ grupos de forma que cada um dos k grupos não pode ter mais do que $\lfloor n\epsilon^2 \rfloor = q$ objetos. Vamos chamar cada grupo dessa partição de G_i, onde $i \in \{1, \ldots, k\}$ indexa os grupos, sendo que G_1 é o grupo dos menores objetos e G_k o grupo dos maiores.

Vamos construir uma nova instância para o problema do empacotamento chamada de J. Essa instância possui os mesmos n objetos de I, porém o tamanho de cada objeto $j \in G_i$ é igual ao tamanho do maior objeto em G_i. Perceba que, dessa forma, J é uma instância com objetos de no máximo k tamanhos diferentes, sendo que todos os objetos possuem tamanho maior ou igual a ϵ, um valor positivo. Então segundo o Lema 3.1 existe algoritmo polinomial que nos dá um empacotamento ótimo sobre J.

Observe que qualquer solução para J pode ser facilmente transformada em um solução para I, porque cada objeto de I cabe no seu correspondente em J.

Vamos construir uma outra instância para o empacotamento chamada L. Essa instância também possui os n objetos de I, porém, o tamanho de cada objeto $l \in G_i$ é igual ao do menor objeto pertencente a G_i. Dessa forma, é evidente que

$$OPT_{Emp}(L) \leq OPT_{Emp}(I).$$

Veja que para as instâncias L e J cada grupo G_i contém objetos de apenas um tamanho. Dito isso, vamos denotar por $G_i(J)$ o tamanho dos objetos do i-ésimo grupo da instância J e por $G_i(L)$ o tamanho dos objetos do i-ésimo grupo da instância L. Observe que $\forall i \in \{1, \ldots, k-1\}$ vale que $G_i(J) \leq G_{i+1}(L)$. Então, um empacotamento para J que

- aloque os objetos pertencentes aos $k-1$ primeiros grupos da mesma forma que em um empacotamento ótimo para L, e
- empacote cada um dos até q objetos de $G_k(J)$ em um pacote próprio,

é uma solução viável para a instância J, assim

$$OPT_{Emp}(J) \leq OPT_{Emp}(L) + q \leq OPT_{Emp}(I) + q.$$

Veja que, por hipótese, todo elemento possui tamanho maior do que ϵ e como temos n objetos, então

$$n\epsilon \leq OPT(I)_{Emp}.$$

Mas lembre-se que $q = \lfloor n\epsilon^2 \rfloor$, logo $q \leq n\epsilon^2$, o que em conjunto com as desigualdades acima nos leva a

$$q \leq OPT_{Emp}(I)\epsilon \Rightarrow OPT_{Emp}(J) \leq (1+\epsilon)OPT_{Emp}(I).$$

Dessa forma, sabemos que existe um algoritmo polinomial que encontra $OPT_{Emp}(J)$ e também provamos que tal algoritmo é uma $(1+\epsilon)$ aproximação para o Empacotamento em I. Assim concluímos nossa demonstração.

Como já comentado, a solução ótima para J é facilmente convertida em uma solução S para I com valor igual a $OPT_{Emp}(J)$. Logo, o algoritmo polinomial do Lema 3.1 é uma $(1+\epsilon)$-aproximação para o problema do empacotamento.

□

Enunciados esses lemas vamos apresentar o esquema de aproximação que usaremos. Ele começa retirando todos os objetos da instância com tamanho menor ou igual a um determinado ϵ e usa diferentes algoritmos para diferentes partes da instância. Ele primeiro usa o algoritmo previsto no Lema 3.1 que resolve de maneira ótima instâncias restritas. Depois disso ele usa o *First-Fit*, que conforme demonstrado, é uma 2-aproximação.

Algoritmo 17: EMPACOTAMENTO(n, t, ϵ)

Entrada: Quantidade n de objetos disponíveis e função tamanho t
Saída: Conjunto de pacotes P

1 Tome $\epsilon > 0$
2 Construa uma instância J para o Empacotamento usando os objetos de tamanho $\geq \epsilon$ como descrito no Lema 3.2
3 Use o Algoritmo do Lema 3.1 para a instância J que devolve um empacotamento R
4 Em R, devolva seus objetos ao tamanho original obtendo o empacotamento P
5 Empacote os outros objetos de tamanho $\leq \epsilon$ em P usando o *First-Fit*
6 **retorna** P

Segue o teorema que demonstra o fator de aproximação do Algoritmo 17.

Teorema 3.4. *Dada uma instância $I = \langle n, t\rangle$ para o empacotamento. Para todo ϵ, onde $0 < \epsilon < 1/2$, existe um algoritmo $\mathcal{A}_\epsilon$ que possui tempo polinomial em n e encontra um empacotamento com até $(1+2\epsilon)OPT_{Emp}(I)+1$ pacotes.*

Demonstração. Vamos denotar por $I_{\geq\epsilon}$ a instância obtida quando descartamos todos os objetos com valor menor que ϵ. Também considere que R e P tem o mesmo significado que no Algoritmo 17.

Pelo Lema 3.2 sabemos que o empacotamento R possui no máximo $(1+\epsilon)OPT_{Emp}(I_{\geq\epsilon})$ pacotes. Veja que R já considera todos os objetos com tamanho $\geq \epsilon$ da instância I, logo isso também vale para P na linha 4.

Obtido P, quando começarmos a empacotar os objetos de I que possuem tamanho menor do que ϵ pelo *First-Fit* podemos ter duas situações:

1. Não é necessário criar nenhum novo pacote, então temos que

$$|P| = (1+\epsilon)OPT_{Emp}(J) \leq (1+\epsilon)OPT_{Emp}(I).$$

2. Se for necessário criar novos pacotes é evidente que todos os pacotes exceto o último têm no mínimo tamanho ocupado de $1-\epsilon$, ou seja, sabemos que ao final do algoritmo temos no mínimo $|P|-1$ pacotes preenchidos até $1-\epsilon$ (veja a demonstração do Teorema 3.3). Assim,

$$(|P|-1)(1-\epsilon) \leq OPT_{Emp}(I) \Rightarrow |P| \leq \frac{OPT_{Emp}(I)}{(1-\epsilon)} + 1$$

   Por hipótese, sabemos que $0 < \epsilon \leq 1/2$, logo

$$|P| \leq (1+2\epsilon)OPT_{Emp}(I) + 1.$$

Em ambos os casos temos um algoritmo polinomial que nos devolve uma solução com fator de aproximação de até $(1+2\epsilon)OPT_{Emp}(I)+1$.

□

Capítulo 4

Programação Linear

Uma das principais ferramentas usadas no desenvolvimento de algoritmos para problemas de otimização é o método de otimização chamado de ***programação linear*** (PL). Boa parte da teoria sobre algoritmos de aproximação é construída envolta de PL [6].

A PL usa problemas de otimização específicos que visam otimizar o valor de uma dada função linear, sendo que existe um conjunto de restrições que as variáveis dessa função devem respeitar. Esses problemas de otimização são chamados de ***programas lineares*** e podem ser de minimização ou de maximização.

Observe que chamamos de programação linear o método de otimização em si, já o problema de minimização (ou de maximização) é chamado de ***programa linear*** (PL[1]). A função que desejamos minimizar (ou maximizar) é chamada de ***função objetivo*** e o sistema de equações e inequações que dá as restrições das variáveis é chamado de ***conjunto de restrições***. Segue abaixo o enunciado do programa linear de minimização.

Problema 4.1 (Programa Linear de Minimização)**.** *Dados uma matriz* $A = (a_{ij}) \in \mathbb{Q}^{m \times n}$*, um vetor* $c = (c_i) \in \mathbb{Q}^n$ *e um vetor* $b = (b_i) \in \mathbb{Q}^m$*, desejamos*

[1]Vamos usar a sigla PL tanto para nos referirmos à Programação Linear quanto a um Programa Linear.

encontrar um vetor $x = (x_i) \in \mathbb{Q}^n$ *que minimize a seguinte função*

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n,$$

ao mesmo tempo que respeite as seguintes condições

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \geq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \geq b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \geq b_m \end{cases}$$

Como o próprio nome já indica, tanto a função objetivo quanto as restrições devem ser lineares em relação às variáveis do vetor x. O conjunto de todos os vetores x que respeitam as restrições do programa é chamado de ***poliedro*** do programa. Cada vetor que o compõe o poliedro é uma ***solução viável***. As soluções de um programa linear são chamadas de ***soluções ótimas***. Cada solução viável também é chamada de ***ponto*** do poliedro. Segue abaixo um exemplo de programa linear de minimização.

Exemplo 4.1. *Desejamos encontrar os valores de* $x_1, x_2 \in \mathbb{Q}$ *que minimizem o valor de*

$$-2x_1 - x_2$$

Ao mesmo tempo que respeitem

$$\begin{cases} x_1 + x_2 \leq 5 \\ 2x_1 + 3x_2 \leq 12 \\ x_1 \leq 4 \\ x_1 \geq 0 \\ x_2 \geq 0 \end{cases}$$

O poliedro desse programa linear é representado pela área hachurada na Figura 4.1. Também veja que cada restrição do programa é representada por

uma reta delimitante dessa área na Figura 4.1.

A definição do programa linear de minimização, bem como o de maximização, é de certa forma genérica. Graças a isso podemos reduzir muitos outros problemas de otimização a programas lineares de maneira bem intuitiva. Dada essa praticidade que programas lineares nos oferecem, é possível perceber que dados a redução de algum problema de otimização para um programa linear e um método que resolve programas lineares no geral, então conseguimos encontrar a solução do problema reduzido.

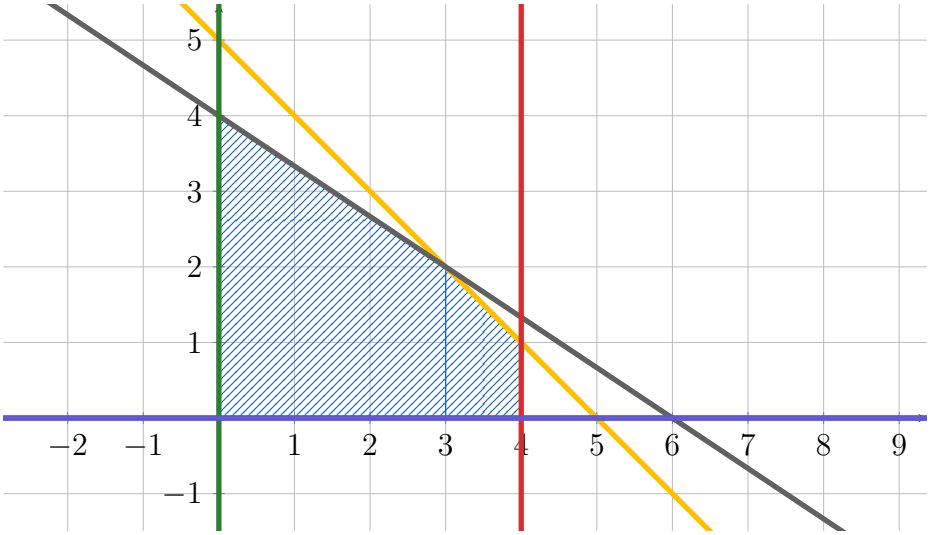


Figura 4.1: Representação do poliedro do PL do Exemplo 4.1.

De fato, para problemas com resolução em tempo polinomial é possível aplicar amplamente a PL, de forma a obter algoritmos para a resolução desses problemas [6]. Porém, devemos nos perguntar se também podemos aproveitar a PL para resolver problemas que sejam $\mathcal{NP}$-difíceis.

Para trabalhar com problemas discretos $\mathcal{NP}$-difíceis não usamos PL, mas sim a ***programação linear inteira*** (PLI), sendo que a grande mudança é a de que agora as variáveis do programa só podem assumir valores inteiros. Segue abaixo um exemplo em que descrevemos o problema da mochila

(Problema 3.1) através de um programa linear inteiro de maximização.

Exemplo 4.2. *Dados um valor $n \in \mathbb{Z}$, uma função $v : \{1, 2, \ldots, n\} \to \mathbb{Z}$, uma função $s : \{1, 2, \ldots, n\} \to \mathbb{Z}$, um valor $B \in \mathbb{Z}$ e um vetor $x = (x_i) \in \mathbb{Z}^n$, desejamos encontrar x para*

$$\begin{aligned} \textit{maximizar} \quad & \sum_{i=1}^{n} v_i x_i \\ \textit{sujeito a} \quad & \sum_{i=1}^{n} s_i x_i \leq B \\ & x_i \in \{0, 1\} \qquad\qquad 1 \leq i \leq n \end{aligned}$$

Veja que acabamos de reduzir um problema $\mathcal{NP}$-difícil para um programa linear inteiro, e é justamente por causa disso que, por mais que pareça uma mudança pequena, existem grandes diferenças computacionais entre um PL e um PLI. Segue o teorema que demonstra isso .

Teorema 4.1. *[4] Um programa linear inteiro é $\mathcal{NP}$-difícil.*

Ao modelarmos um problema $\mathcal{NP}$-difícil por PLI, geralmente usamos duas estratégias:

- Modelagem por variáveis binárias, onde uma variável do programa linear só pode assumir valores 0 ou 1, sendo normalmente usada em problemas onde temos que decidir se um objeto faz ou não parte da solução, como no problema da Mochila;

- Modelagem por variáveis inteiras, onde a variável pode assumir valores inteiros não necessariamente binários, um exemplo seria o programa linear para o Empacotamento (Problema 3.2).

4.1 Dualidade

Na teoria da PL temos o **teorema da dualidade** que relaciona um programa linear de minimização com um problema linear de maximização. Para entender a ideia por trás desse teorema, vamos usar ao longo dessa seção o seguinte exemplo de programa linear de minimização.

Exemplo 4.3. *Seja* $x = (x_1, x_2, x_3) \in \mathbb{Q}^3$. *Desejamos*

$$\begin{aligned} \textit{minimizar} \quad & 7x_1 + x_2 + 5x_3 \\ \textit{sujeito a} \quad & x_1 - x_2 + 3x_3 \geq 10 \\ & 5x_1 + 2x_2 - x_3 \geq 6 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Veja que cada valor de x que respeite as condições impostas é uma solução viável do PL. Vamos denotar por x^* a solução ótima desse programa e por $f(x^*)$ o valor dessa solução ótima. Agora, dado um $\alpha \in \mathbb{Q}$ será verdade que $f(x^*) \leq \alpha$?

Para obtermos uma resposta "sim" para essa pergunta basta encontrarmos uma solução viável x qualquer tal que $f(x) = \alpha$. Por exemplo, seja $\alpha = 30$, como $x = (2, 1, 3)$ é solução viável e $f(x) = 30$, então $f(x^*) \leq 30$, pois esse é um problema de minimização.

Agora, como podemos obter uma resposta "não" para aquela pergunta? Observe que, se conseguirmos encontrar algum valor α tal que $\alpha \leq f(x^*)$, então encontramos um limitante inferior para o valor ótimo do problema. Nossa primeira ação pode ser olhar as restrições do programa. No Exemplo 4.3, $x_1, x_2, x_3 \geq 0$, logo,

$$7x_1 + x_2 + 5x_3 \geq x_1 - x_2 + 3x_3 \geq 10.$$

Assim, podemos concluir que qualquer solução viável possui como limitante inferior o valor 10. Se somarmos as duas restrições que nos foram dadas

obtemos um limitante ainda mais justo, pois

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 16.$$

A expressão acima sugere que uma combinação linear das restrições pode fornecer um limitante inferior melhor para o problema. O mais interessante nesse processo é que podemos escrever um PL de maximização com base na combinação linear das restrições. Para enxergamos isso, vamos desenvolver as seguintes inequações, para $y_1, y_2 \in \mathbb{Z}$:

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3)y_1 + (5x_1 + 2x_2 - x_3)y_2 \geq 10y_1 + 6y_2$$

$$7x_1 + x_2 + 5x_3 \geq x_1y_1 - x_2y_1 + 3x_3y_1 + 5x_1y_2 + 2x_2y_2 - x_3y_2 \geq 10y_1 + 6y_2$$

$$7x_1 + x_2 + 5x_3 \geq x_1(y_1 + 5y_2) + x_2(-y_1 + 2y_2) + x_3(3y_1 - y_2) \geq 10y_1 + 6y_2$$

As inequações acima resultam no seguinte PL de maximização.

Exemplo 4.4. *Seja $y = (y_1, y_2) \in \mathbb{Q}^2$. Desejamos*

$$\begin{aligned} \textit{maximizar} \quad & 10y_1 + 6y_2 \\ \textit{sujeito a} \quad & y_1 + 5y_2 \leq 7 \\ & -y_1 + 2y_2 \leq 1 \\ & 3y_1 - y_2 \leq 5 \\ & y_1, y_2 \geq 0 \end{aligned}$$

Veja que encontramos um PL de maximização (Exemplo 4.4) a partir de um PL de minimização (Exemplo 4.3). Quando fazemos esse tipo de construção relacionando PLs sempre chamamos o primeiro programa de ***programa primal*** e o segundo de ***programa dual***. O programa dual de um programa dual é sempre o próprio programa primal. Por construção, também podemos notar que qualquer solução viável do dual é sempre um limitante de otimização para o programa primal, sendo que a inversa também vale.

Um outro fato muito importante é que se encontrarmos uma solução para o primal e uma solução para o dual que possuam o mesmo valor, então esse é o valor da solução ótima para os dois PLs. Essa propriedade dos PLs que exemplificamos através de nossos exemplos é resultado do teorema central da Programação Linear [6]. Antes de enunciarmos esse teorema, dado o Programa Linear de Minimização (Problema 4.1), considere agora seu dual, o seguinte problema de maximização.

Problema 4.2 (Programa Linear de Maximização)**.** *Dadas uma matriz* $A = (a_{ij}) \in \mathbb{Q}^{n\times m}$*, um vetor* $c = (c_i) \in \mathbb{Q}^n$ *e um vetor* $b = (b_i) \in \mathbb{Q}^m$*, desejamos encontrar um vetor* $y = (y_i) \in \mathbb{Q}^m$ *que maximize a seguinte função*

$$b_1y_1 + b_2y_2 + \cdots + b_my_m$$

Ao mesmo tempo que respeite as seguintes condições

$$\begin{cases} a_{11}y_1 + a_{12}y_2 + \cdots + a_{1m}y_n \leq c_1 \\ a_{21}y_1 + a_{22}y_2 + \cdots + a_{2m}y_n \leq c_2 \\ \vdots \\ a_{n1}y_1 + a_{n2}y_2 + \cdots + a_{nm}y_n \leq c_n \end{cases}$$

Teorema 4.2 (Teorema da Dualidade [6])**.** *Dados o Programa Linear de Minimização como primal e o Programa Linear de Maximização como dual, se as variáveis das funções objetivo de ambos só podem assumir valores positivos, então o programa primal possui uma solução ótima finita se, e somente se, seu programa dual também tem solução ótima finita. Mais do que isso, dada uma solução ótima para o primal* $x^* = (x_1, x_2, \ldots, x_n)$ *e dada uma solução ótima para o seu dual* $y^* = (y_1, y_2, \ldots, y_m)$*, então*

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*.$$

Ao enunciarmos o Teorema 4.2 usamos os Problemas 4.1 e 4.2, pois ambos são programas genéricos, no sentido de que podemos escrever qualquer programa de minimização ou maximização usando um dos dois como modelo. Além disso, ao enunciarmos o Teorema 4.2 utilizamos como primal o problema de minimização, mas não faz diferença qual seja o programa primal, esse teorema é sempre válido. Como já falamos, o dual do dual é o próprio primal.

Voltando um pouco, vimos que dado um problema primal de minimização, qualquer solução viável de seu programa dual dá um limitante inferior para a solução ótima. Esse resultado é interessante, mas não é tão forte quanto o Teorema da Dualidade Fraca e acaba sendo chamado de Teorema da Dualidade Fraco. Enquanto muitos algoritmos exatos possuem seu projeto baseados no Teorema da Dualidade. Para algoritmos de aproximação muitas vezes apenas o Teorema da Dualidade Fraca é suficiente .

Teorema 4.3 (Teorema da Dualidade Fraca)**.** *[6] Dados o Programa Linear de Minimização como primal e o Programa Linear de Maximização como dual, se* $x = (x_1, x_2, \ldots, x_n)$ *é solução viável para o primal e* $y = (y_1, y_2, \ldots, y_m)$ *é solução ótima para o dual, então*

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i.$$

Demonstração. Como y é solução viável do dual e todo x_j é não negativo, pelas restrições do dual obtemos que

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j.$$

Da mesma forma, como x é solução viável do primal e todo y_j é não

negativo, pelas restrições do primal obtemos que

$$\sum_{i=1}^{m} b_i y_i \leq \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i.$$

Mas veja que a seguinte igualdade é válida

$$\sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i.$$

Então, concluímos que

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i.$$

□

Veja que o Teorema da Dualidade só é válido quando no Teorema da Dualidade Fraca temos uma igualdade. Caso isso aconteça obtemos o que chamamos de Condições de Folga Complementares, que são extremamente importantes para o projeto de algoritmos eficientes, sejam exatos ou de aproximação [6]. Segue o teorema que enuncia o que são essas condições.

Teorema 4.4 (Condições de Folga Complementares)**.** *[6] Dados o Programa Linear de Minimização como primal e o Programa Linear de Maximização como dual, se $x = (x_1, x_2, \ldots, x_n)$ é solução viável para o primal e $y = (y_1, y_2, \ldots, y_m)$ é solução ótima para o dual, então x e y são soluções ótimas se, e somente se, as seguintes condições são satisfeitas:*

- ***Folga Primal**: Para cada $1 \leq j \leq n$, $x_j = 0$ ou $\sum_{i=1}^{m} a_{ij} y_i = c_j$; e*
- ***Folga Dual**: Para cada $1 \leq i \leq m$, $y_i = 0$ ou $\sum_{j=1}^{n} a_{ij} x_j = b_i$.*

O que o resultado acima nos diz é que dadas duas soluções ótimas x e y para um primal e um dual respectivamente, sendo que ambas são vetores,

então ou cada variável de cada uma delas assume valor 0 ou as condições de restrição do problema também assumem valor de igualdade.

4.2 Relaxação

Dados dois PLs, PA e PB, que possuem a mesma função objetivo, nós falamos que PB é **relaxação** de PA, se qualquer solução viável de PA é solução viável de PB, mas o inverso não é necessariamente verdade. Para obtermos uma relaxação de um determinado PL nós excluímos algumas de suas restrições, o que leva a um aumento do poliedro de soluções viáveis, porém, obtemos um outro PL que é uma relaxação.

De maneira geral, uma relaxação é um programa linear que escrevemos a partir de outro, porém, conforme o próprio nome indica, as restrições do novo programa são mais permissivas. Relaxações geralmente são usadas, pois às vezes conseguimos usar uma solução viável de uma relaxação para encontrar uma boa solução viável do programa original. Além disso, é evidente que qualquer solução viável de uma relaxação é um limitante para a solução ótima do programa, o que é verdade pois o domínio de soluções viáveis da relaxação inclui todas as soluções viáveis do PLI.

Quando a relaxação de um PLI consiste em desconsiderar as restrições de integralidade das variáveis do problema, chamamos essa relaxação de ***relaxação linear*** do PLI. Perceba que o poliedro de qualquer relaxação linear é infinitamente maior que as soluções viáveis de um PLI. No Exemplo 4.1 temos um PL que poderia muito bem representar a relaxação linear de um PLI que tivesse a mesma função objetivo e as mesmas restrições. Na Figura 4.2 representamos através de pontos as soluções viáveis do PLI para o qual o Exemplo 4.1 seria a relaxação linear, veja a diferença na quantidade soluções viáveis entre os dois.

Caso a solução ótima da relaxação seja composta só por valores inteiros, então essa solução também é ótima para o PLI. Isso é verdade por causa

Figura 4.2: Representação das soluções viáveis do PLI cuja relaxação linear é o Exemplo 4.1.

da relação dos domínios de soluções viáveis dos dois, afinal com certeza não existe solução viável para o PLI que otimize mais a função objetivo, senão tal solução seria a solução ótima também da relaxação.

4.3 Algoritmos e PL

Conforme já falamos, podemos modelar diversos problemas de otimização combinatória através de PLI, mas ainda não chegamos a falar como a modelagem por PLI auxilia na construção de algoritmos. Feita a modelagem de um problema por PLI, a relaxação linear desse PLI nos dá uma excelente forma de encontrar limitantes de otimização para o problema original. Mas veja que a relaxação linear de um problema é sempre um PL e, diferente de um PLI, existem bons algoritmos que nos permitem encontrar sua solução ótima [2]. Então, nós aproveitamos da existência desses algoritmos eficientes para PLs e, de alguma forma, encontramos soluções exatas ou aproximadas

para o PLI e, consequentemente, para o problema original.

Existem muitos detalhes e peculiaridades de cada algoritmo existente para resolver PLs, porém, não cabe a esse projeto fazer isso, afinal, estamos interessados no uso de PLs para projetar de algoritmos de aproximação. Dessa forma, vamos nos limitar à uma breve descrição dos principais algoritmos existentes.

O mais famoso algoritmo usado para resolver PLs é o ***algoritmo simplex***, que resolve qualquer par dual de programas lineares. Ao receber um primal e um dual o simplex decide se os dois problemas são viáveis e, em caso afirmativo, ele produz soluções ótimas, dentro dos racionais, para ambos os problemas [2]. O funcionamento do simplex consiste em usar um método numérico iterativo. O simplex é em média eficiente, mas não é limitado por uma função polinomial. Existem algoritmos muito diferentes do simplex que são capazes de resolver PLs em tempo polinomial como, por exemplo, o ***algoritmo das elipsoides***. Mesmo com a existência desses algoritmos eficientes, o método simplex costuma ser o mais utilizado para resolver PLs.

4.4 Algoritmos de Aproximação e PL

Agora que temos uma visão geral sobre a relação entre o projeto de algoritmos e a programação linear, vamos falar sobre o uso da programação linear para obtenção de algoritmos de aproximação. Segundo Vazirani [6], existem duas técnicas que são as mais utilizadas para obter algoritmos de aproximação de um PLI. A primeira e mais óbvia consiste em encontrar a solução ótima da relaxação em que estamos trabalhando e converter tal solução em uma boa solução viável para o PLI, técnica conhecida como ***arredondamento***. A outra técnica, menos óbvia e mais sofisticada, trabalha com uma sequência de soluções das relaxações do primal e do dual para, de maneira iterativa, ir estabelecendo limitantes para a solução devolvida, técnica conhecida como ***esquema primal-dual***.

Veja que o arredondamento é uma técnica que usa somente a relaxação linear do programa para obter a relação aproximativa, o que a caracteriza como uma técnica primal. Já o esquema primal-dual, como o próprio nome sugere, constrói o fator de aproximação baseando-se nas relações existentes entre a relaxação linear do programa e seu dual, sendo assim uma técnica que usa os dois programas.

Para nos aprofundarmos um pouco mais na teoria utilizada para construção de algoritmos de aproximação usando PLs, vamos apresentar um conceito importante que será usado para estabelecer o melhor fator de aproximação que podemos esperar encontrar para qualquer algoritmo de aproximação.

Definição 4.1 (Gap de Integralidade)**.** *Dado um problema de minimização A, que possa ser escrito através de um PLI, denotaremos por B a relaxação linear do PLI de A. Vamos denotar por $OPT_A(I)$ a solução ótima de A, logo solução ótima de seu PLI, e vamos denotar por $OPT_B(I)$ a solução ótima de sua relaxação linear. Seja I uma instância para A. A razão*

$$\frac{OPT_A(I)}{OPT_B(I)},$$

é chamada ***gap de integralidade*** *para a instância I.*

No caso de um problema de maximização, o gap de integralidade é o inverso da razão apresentada acima. Dessa forma, tanto para problemas de minimização quanto para os maximização, o valor do gap de integralidade é maior ou igual a 1. O motivo de termos definido esse novo conceito é a apresentação do seguinte teorema.

Teorema 4.5. *Dados um problema de otimização combinatória e uma instância I para tal problema. Qualquer algoritmo de aproximação que tenha sido projetado para determinado problema, usando a relaxação linear do problema, não pode alcançar um fator de aproximação melhor, para a instância I, do que a lacuna de integralidade do problema para essa instância [6].*

Para a maioria dos problemas, tanto a técnica de arredondamento quanto o esquema primal-dual fornecem bons algoritmos que conseguem atingir um fator de aproximação basicamente igual ao valor do gap de integralidade [6].

A principal diferença entre as técnicas que citamos é o tempo de execução dos algoritmos produzidos. Geralmente, algoritmos que usam o arredondamento limitam-se a resolver o PL primal e fazer o processo de arredondamento, o que pode levar a tempos de execução exorbitantes dependendo do PL que deve ser resolvido. Já algoritmos que usem primal-dual conseguem aproveitar-se das estruturas combinatórias do problema e de subproblemas que possam vir a ser usados, possuindo um tempo de consumo geralmente menor que o de algoritmos por arredondamento.

Capítulo 5

Algoritmos Probabilísticos e Satisfatibilidade Máxima

Nesse capítulo falaremos sobre algoritmos probabilísticos e sua relação com algoritmos de aproximação. Começaremos definindo conceitos importantes para a teoria de algoritmos probabilísticos e depois disso apresentaremos o problema da Satisfatibilidade Máxima e três algoritmos de aproximação probabilísticos para o mesmo. Também é importante destacar que um desses algoritmos usará programação linear. Os conteúdos apresentados nesse capítulo foram retirados de Carvalho et al. [2] e Vazirani [6].

5.1 Algoritmos Probabilísticos

Nessa seção apresentaremos alguns conceitos relacionados a algoritmos probabilísticos cujo conhecimento é importante para que entendamos o uso e funcionamento desse tipo de algoritmo. Basicamente apresentaremos diversas definições e algum detalhamento da relação entre algoritmos de aproximação e algoritmos probabilísticos. Vamos começar relembrando de algumas definições básicas da teoria das probabilidades.

Definição 5.1 (Experimento Aleatório)**.** *Entendemos por* **experimento aleatório** *qualquer fenômeno que possua resultados imprevisíveis mesmo se repetido nas mesmas condições.*

Definição 5.2 (Espaço Amostral)**.** *Dado um experimento aleatório, chamamos de* **espaço amostral** *o conjunto de todos os resultados possíveis desse experimento. Denotamos o espaço amostral de um experimento aleatório por* Ω*.*

Definição 5.3 (Medida Discreta de Probabilidade)**.** *Uma* **medida discreta de probabilidade** *sobre um espaço amostral* Ω *finito e não vazio é uma função* $P : \Omega \to [0, 1]$ *que respeita a seguinte igualdade:*

$$P(\Omega) := \sum_{\omega \in \Omega} P(\omega) = 1$$

Definição 5.4 (Espaço Discreto de Probabilidade)**.** *Seja* P *uma medida discreta de probabilidade sobre um conjunto* Ω*, dizemos que o par* (Ω, P) *é um* **espaço discreto de probabilidade***.*

Definição 5.5 (Variável Aleatória Discreta)**.** *Uma* **variável aleatória** *sobre um conjunto* Ω *finito e não vazio é uma função* $X : \Omega \to \mathbb{R}$*.*

Definição 5.6 (Evento)**.** *Dada uma variável aleatória* X *sobre um espaço amostral* Ω *e um conjunto* $S \subseteq Img[X]$*, chamamos* S *de* **evento***. Representamos um evento* S *por* $[X \in S]$ *ou, caso* $S = \{x\}$*, por* $[X = x]$*.*

Definição 5.7 (Probabilidade de um Evento)**.** *Sejam uma variável aleatória discreta* X *em* Ω*, um evento* $S \in Img[X]$ *e uma medida discreta de probabilidade* P *em* Ω*. Dizemos que a* **probabilidade do evento** $[X \in S]$ *é dada pela soma dos valores* $P(\omega)$*, onde* ω *é todo elemento de* Ω *tal que* $X(\omega) \in S$*. Representamos a probabilidade desse evento por* $Pr[X \in S]$*.*

Definição 5.8 (Esperança Matemática)**.** *A* **esperança matemática** *de uma variável aleatória discreta X é definida como*

$$E[X] := \sum_{x \in S} xP[X = x].$$

Apresentados todos esses conceitos segue um exemplo que exemplifica o usa praticamente todos esses conceitos de maneira bem intuitiva.

Exemplo 5.1. *Vamos considerar o experimento aleatório de lançar duas vezes uma moeda. Sabemos que a cada lançamento só podemos ter dois resultados, cara (c) ou coroa (c′), e dessa forma já sabemos qual é o nosso espaço amostral,* $\Omega = \{cc, cc', c'c, c'c'\}$.

Agora, como poderíamos relacionar esse experimento aleatório com variáveis aleatórias? Para fazer isso vamos definir uma variável aleatória X que responda "Quantas caras obtivemos nesse evento?". Veja que $X(0) = 1, X(1) = 2$ *e* $X(2) = 1$.

Veja que devido à forma como nossa variável aleatória foi definida, os resultados $c'c$ e cc' são equivalentes, representado apenas um resultado.

Considere a definição da seguinte função de probabilidade $P[X = x] = X(x)/|\Omega|$. *Logo,* $P[X = 0] = \frac{1}{4}$, $P[x = 1] = \frac{1}{2}$. *Dessa forma, a esperança matemática dessa variável aleatória é dada por:*

$$E[X] = 0\frac{1}{4} + 1\frac{1}{2} + 2\frac{1}{4} = 1$$

Agora, vamos começar a falar sobre conceitos relacionados a algoritmos probabilísticos.

Definição 5.9 (Gerador de Aleatório Bits)**.** *Considere que tenhamos um algoritmo chamado* RAND *e que ele receba como entrada um valor racional p que esteja no intervalo* $[0, 1]$. RAND *devolve como saída o valor* 1 *com probabilidade p ou devolve o valor* 0 *com probabilidade* $1 - p$. *Qualquer algoritmo que possuir um comportamento igual ao de* RAND *é chamado de*

gerador de aleatório de bits.

Ao longo desse capítulo, quando nos referirmos a RAND estaremos falando de um gerador aleatório de bits.

Definição 5.10 (Algoritmo Probabilístico)**.** *Se um algoritmo usa um gerador aleatório de bits, seu comportamento não depende apenas da instância de entrada, mas também dos valores devolvidos por* RAND*. Devido a isso, chamamos esse tipo de algoritmo de* **algoritmo probabilístico**.

Apresentado o que é um algoritmo probabilístico vamos definir o que é ser um algoritmos probabilístico polinomial.

Definição 5.11 (Algoritmo Probabilístico Polinomial)**.** *Dado um algoritmo probabilístico que resolva um problema H, dizemos que esse algoritmo é um* **algoritmo probabilístico polinomial** *se para toda instância I de H existe um polinômio $p(|I|)$, onde $|I|$ é o tamanho da palavra de I, que respeite as seguintes condições:*

1. *O número de chamadas de* RAND *feitas pelo algoritmo é $O(p(|I|))$; e*
2. *O consumo de tempo das operações no algoritmo não relacionadas a* RAND *é limitado por $O(p(|I|))$.*

Agora que não só apresentamos o que é um algoritmo probabilístico como também apresentamos as condições para que esse tipo de algoritmo seja polinomial, vamos apresentar uma definição que faz a ponte entre algoritmos probabilísticos e algoritmos de aproximação.

Definição 5.12 (Algoritmo de Aproximação Probabilístico e Razão de Aproximação Esperada)**.** *Considere um algoritmo probabilístico A para um problema de otimização combinatória H. Para qualquer instância I de H, vamos representar por X_I a variável aleatória que possua o mesmo valor da solução produzida por A para I. Dizemos que A é um* ***algoritmo de aproximação probabilístico*** *com* ***razão de aproximação esperada*** *α, se:*

- $E[X_I] \geq \alpha OPT_H(I)$, *caso* H *seja de maximização; ou*
- $E[X_I] \leq \alpha OPT_H(I)$, *caso* H *seja de minimização;*

Também dizemos que A *é uma* α**-aproximação probabilística**.

Na definição acima não apresentamos α como dependente de I, porém, isso é uma situação que pode ocorrer. Por mais que a esperança do resultado devolvido por um algoritmo de aproximação probabilístico seja boa, podemos provar algo ainda melhor. Vamos agora apresentar alguns argumentos que provam que, dadas certas condições, uma α-aproximação probabilística produz, com alta probabilidade, uma solução aproximada do problema. Antes de apresentarmos esse resultado vamos enunciar a desigualdade de Markov.

Teorema 5.1 (Desigualdade de Markov)**.** *Dados um espaço discreto de probabilidade* (Ω, P)*, uma variável aleatória* X *sobre* Ω*, onde todo valor de* X *é não negativo, e dado também um valor não negativo* λ*, então*

$$P[X \geq \lambda] \leq \frac{E[X]}{\lambda}.$$

Agora, considere que temos uma α-aproximação probabilística A para um problema de minimização, e uma variável aleatória X_I, relacionada à saída de A para cada instância I do problema, que só assume valores positivos. Sabemos que por A ser uma α-aproximação vale que

$$E[X_I] \geq \alpha OPT_H(I).$$

Agora, pela Desigualdade de Markov apresentada, sabemos que para cada $\epsilon > 0$ também é válido que

$$P[X_I \geq (\alpha + \epsilon)OPT_H(I)] \leq \frac{E[X_I]}{(\alpha + \epsilon)OPT_H(I)} \leq \frac{\alpha}{\alpha + \epsilon},$$

Sendo que a segunda desigualdade é válida justamente pela definição de uma α-aproximação probabilística.

Vamos definir $k = \lceil \log_\beta \lambda \rceil$, onde $\beta = \frac{\alpha}{\alpha+\epsilon}$. Considere que nós executamos nosso algoritmo k vezes sobre a instância I e que ele devolve ao final o menor dos valores encontrados. Vamos chamar tal valor de Y_I. Veja que Y_I é em si uma variável aleatória derivada do experimento que é repetir a aplicação do algoritmo k vezes obtendo k valores para X_I. Assim, se definimos o espaço de probabilidade de cada X_I como (Ω, P), então o espaço de probabilidade de Y_I é $(\Omega, P)^k$.

Para cada uma das k vezes que rodamos o algoritmo vale que $P[X_I \geq (\alpha+\epsilon)OPT_H(I)] \leq \beta$. Como Y_I é o menor valor assumido por um X_I, para que ele seja maior ou igual a algo, cada X_I tem que ser maior ou igual a esse mesmo algo, ou seja, k eventos envolvendo a escolha dos X_I devem ocorrer juntos. Esses dois últimos argumentos nos levam à seguinte conclusão:

$$P[Y_I \geq (\alpha+\epsilon)OPT_H(I)] \leq \beta^k \leq \lambda.$$

Assim, o algoritmo A devolve uma solução com valor menor ou igual a $(\alpha+\epsilon)OPT_H(I)$ com probabilidade de pelo menos $(1-\lambda)$, o que é especialmente bom quando λ é muito pequeno.

Para encerrar a seção vamos falar de duas classes de problemas de decisão que envolvem em sua definição algoritmos probabilísticos. Ambas as definições foram retiradas de Trevisan [5].

Definição 5.13 (Classe $\mathcal{BPP}$)**.** *Seja P um problema de decisão. Dizemos que P faz parte da classe $\mathcal{BPP}$ se P pode ser resolvido por um algoritmo probabilístico polinomial tal que para toda instância de P a probabilidade de erro é no máximo 1/3.*

Definição 5.14 (Classe $\mathcal{RP}$)**.** *Seja P um problema de decisão. Dizemos que P faz parte da classe $\mathcal{RP}$ se P pode ser resolvido por um algoritmo probabilístico polinomial tal que para toda instância de P cuja resposta é* NÃO *a probabilidade de acerto é 1 e para toda instância de P cuja resposta é* SIM *a chance de erro é no máximo $1/2$.*

Definição 5.15 (Classe $\mathcal{ZPP}$)**.** *Seja P um problema de decisão. Dizemos que P faz parte da classe $\mathcal{ZPP}$ se P pode ser resolvido por um algoritmo probabilístico em tempo polinomial tal que para toda instância de P.*

$\mathcal{BPP}$ significa *bounded-error probabilistic polynomial time*, $\mathcal{RP}$ significa *randomized polynomial time* e $\mathcal{ZPP}$ significa *zero-error probabilistic polynomial time*.

5.2 Satisfatibilidade Máxima (*Max-Sat*)

Nessa seção apresentaremos o problema da Satisfatibilidade Máxima e alguns algoritmos de aproximação probabilísticos para o mesmo. Vale destacar que o problema da Satisfatibilidade Máxima foi muito importante no desenvolvimento da teoria da complexidade de aproximações [6]. Antes de apresentarmos de fato o problema temos que introduzir alguns conceitos e alguma notação para facilitar o entendimento do mesmo.

Suponha que tenhamos um conjunto finito V de objetos que vamos chamar de ***variáveis***. Dizemos que uma ***cláusula booleana*** sobre V é um par ordenado $\{C_0, C_1\}$ de subconjuntos de V que sejam disjuntos entre si, sendo que ao menos um desses conjuntos não é vazio. Vamos denotar um desses dois conjuntos por C_1, conjunto das variáveis ***não complementadas***, e o outro conjunto por C_0, conjunto das variáveis ***complementadas***. Dizemos que o **número de variáveis** em um cláusula é definido por $|C_1| + |C_0|$.

Uma ***valoração*** das variáveis de V é um vetor x, indexado pelas variáveis $v \in V$, onde cada $x_v \in \{0, 1\}$. Dizemos que uma valoração específica ***satisfaz*** uma cláusula C se para cada $x_v = 1$ existe ao menos um $v \in C_1$ ou se para cada $x_v = 0$ existe ao menos um $v \in C_0$. Então C_0 é o conjunto das variáveis complementadas (valor 0 satisfaz) e que C_1 é o conjunto das variáveis não complementadas (valor 1 satisfaz).

Outra maneira de ver esses conceitos é através de ***expressões booleanas***. Podemos usar expressões booleanas para representar uma cláusula

booleana da seguinte forma. Uma cláusula C que use k variáveis de V pode ser representada como uma expressão booleana $E = \vee_{i=1}^{k} x_i$, onde x é um vetor de valoração de V, sendo que sempre que x satisfaz C também é verdade que $E = 1$. Ao longo dessa seção usaremos essas duas notações, cláusula e expressão.

Exemplo 5.2. *Seja* $V = \{a, b, c, d, e\}$ *e as seguintes cláusulas:* $(\{a, b\}, \{d, e\})$, $(\{a, c\}, \{b, e\})$ *e* $(\{a, d, e\}, \{\emptyset\})$.

Para as cláusulas dadas, um exemplo de valoração seria (0,0,1,1,1). Perceba que essa valoração satisfaz as três cláusulas apresentadas.

Usando notação de expressão booleana podemos representar esse conjunto de cláusulas por $(a' \vee b' \vee d \vee e), (a' \vee b \vee c' \vee e)$ *e* $(a' \vee d' \vee e')$. *Perceba que as valorações que satisfaçam essas cláusulas fazem com que essas expressões booleanas assumam valor lógico* 1.

Apresentadas essas notações e terminologias relacionadas ao problema, segue a descrição formal da Satisfatibilidade Máxima.

Problema 5.1 (Satisfatibilidade Máxima)**.** *Dado um conjunto de variáveis* V *e dada uma coleção* $\mathcal{C}$ *de cláusulas booleanas, desejamos encontrar uma única valoração* x *de* V *que satisfaça o maior número possível de cláusulas.*

Perceba que a entrada do problema consiste em uma coleção $\mathcal{C}$ de cláusulas e em um conjunto de variáveis. Sua saída é uma valoração das variáveis em V feita através de um vetor x, ou seja, uma valoração das variáveis é uma solução viável do problema. Além disso, esse é um problema de maximização onde o que desejamos maximizar é a quantidade de cláusulas cobertas pela nossa valoração resposta.

Infelizmente o problema da Satisfatibilidade Máxima é $\mathcal{NP}$-difícil até mesmo quando cada cláusula em $\mathcal{C}$ tem duas variáveis [2]. Ao longo dessa seção, ao nos referirmos ao Problema 5.1 vamos usar o termo MaxSat.

O primeiro algoritmo probabilístico que veremos para o MaxSat é o Algoritmo de Johnson, que é um algoritmo muito simples e que consegue ser

uma (1/2)-aproximação para o problema. Sua simplicidade é tanta que nem se dá ao trabalho de saber quais as cláusulas em $\mathcal{C}$. Segue o algoritmo.

Algoritmo 18: MaxSat-Johnson(V, $\mathcal{C}$)

Entrada: Conjunto de variáveis V e o conjunto de cláusulas $\mathcal{C}$
Saída: Valoração de variáveis x

```
1 x é o vetor devolvido que representa uma valoração
2 para cada v ∈ V faça
3 |  x_v = RAND(1/2)
4 fim
5 retorna x
```

Não há nem muito o que falar sobre esse algoritmo, mas ele basicamente funciona da seguinte forma. Para cada variável $v \in V$, decidimos se $v = 0$ ou se $v = 1$ jogando uma moeda, que no caso usaria o RAND. Nesse algoritmo o número de chamadas ao RAND, bem como todas as outras operações realizadas é $O(|V|)$.

Vamos definir que $X_{\mathcal{C}}$ é a variável aleatória que devolve a quantidade de cláusulas em $\mathcal{C}$ que são satisfeitas pelo vetor x, devolvido pelo algoritmo 18. O espaço de probabilidade dessa variável aleatória é $(\Omega, P)^{|V|}$, sendo que $\Omega = \{0, 1\}$ e $P(0) = P(1) = \frac{1}{2}$.

Dadas essas informações sobre nosso algoritmo probabilístico, segue o teorema que enuncia seu fator de aproximação.

Teorema 5.2. *Dada uma instância $\langle V, \mathcal{C}\rangle$ para o Problema 5.1, sendo que o número mínimo de variáveis em uma cláusula de $\mathcal{C}$ é k e seja $X_{\mathcal{C}}$ a variável aleatória que representa o número de cláusulas satisfeitas por x devolvido, então*

$$E[X_{\mathcal{C}}] \geq \left(1 - \frac{1}{2^k}\right) OPT_{MS}(V, \mathcal{C}).$$

Demonstração. Para cada cláusula $C \in \mathcal{C}$, vamos definir como Z_C uma variável aleatória que assume valor 1 se C é satisfeita e assume valor 0 caso contrário.

Considere que, dada uma cláusula $C \in \mathcal{C}$, errar a atribuição de uma variável v em C significa atribuir $v = 1$ caso $v \in C_0$ ou atribuir $v = 0$ caso $v \in C_1$.

Tomemos então uma cláusula C_k com menor número de variáveis, ou seja, uma cláusula com k variáveis. Veja que para que essa cláusula não seja satisfeita teríamos que errar a atribuição de todas as suas k variáveis. Porém, para essa cláusula, não interessa a atribuição das outras variáveis de V. Dessa forma, a probabilidade do evento $[Z_{C_k} = 0]$ é dada por $(\frac{1}{2})^k$ e a probabilidade de $[Z_{C_k} = 1]$ é dada por $(1 - \frac{1}{2^k})$.

Por hipótese, C_k é a cláusula com menor número de variáveis, logo $P[Z_{C_k} = 0] \geq P[Z_C = 0]$, isso para todo $C \in \mathcal{C}$. De forma análoga também vale que $P[Z_{C_k} = 1] \leq P[Z_C = 1]$, para todo $C \in \mathcal{C}$.

Assim, podemos concluir que

$$E[X_{\mathcal{C}}] = E\left[\sum_{C \in \mathcal{C}} Z_C\right] = \sum_{C \in \mathcal{C}} E[Z_C] \geq |\mathcal{C}| P[Z_{C_k} = 1] = |\mathcal{C}| \left(1 - \frac{1}{2^k}\right).$$

Veja que o resultado acima prova o teorema, pois $|\mathcal{C}| \geq OPT_{MS}(V, \mathcal{C})$.

□

Como toda cláusula possui ao menos uma variável, podemos usar o Teorema 5.2 para chegar diretamente ao teorema seguinte.

Teorema 5.3. *O Algoritmo 18 é uma* $0,5$*-aproximação probabilística polinomial para o problema de Satisfatibilidade Máxima.*

Vamos agora apresentar um algoritmo um pouco mais sofisticado para a Satisfatibilidade Máxima. Esse próximo algoritmo usa programação linear, mais especificamente, ***arredondamento probabilístico***. Dado um número $p \in (0, 1)$, seu arredondamento probabilístico consiste em arredondar p "para cima" com probabilidade p e "para baixo" com probabilidade $(1 - p)$. Nosso primeiro passo vai ser adaptar a Satisfatibilidade Máxima para programação linear.

Problema 5.2 (PL de Satisfatibilidade Máxima)**.** *Dada uma instância* $\langle V, \mathcal{C} \rangle$*, desejamos encontrar um par de vetores* (x, z)*, onde* x *é indexado por* V *e* z *é indexado por* $\mathcal{C}$*, que*

$$\begin{aligned} \textit{maximize} \quad & \sum_{C \in \mathcal{C}} z_C \\ \textit{sujeito a} \quad & \sum_{v \in C_0} (1 - x_v) + \sum_{v \in C_1} x_v \geq z_C, && \forall C \in \mathcal{C}, \\ & 0 \leq z_C \leq 1, && \forall C \in \mathcal{C}, \\ & 0 \leq x_v \leq 1, && \forall v \in V \end{aligned}$$

Suponha que tenhamos x', uma solução ótima da Satisfatibilidade Máxima para a instância $\langle V, \mathcal{C} \rangle$. Vamos construir um vetor z' da seguinte forma. Para cada cláusula $C \in \mathcal{C}$, se x' satisfaz C, então faça $z'_C = 1$, se não faça $z'_C = 0$. Veja que dessa forma o par (x', z') é uma solução viável para o PL da Satisfatibilidade Máxima, sendo que o valor da função objetivo obtido é a quantidade de cláusulas em $\mathcal{C}$ satisfeitas por x'. Com isso podemos concluir que

$$\sum_{C \in \mathcal{C}} z_C \geq OPT_{MS}(V, \mathcal{C}) = \sum_{C \in \mathcal{C}} z'_C,$$

Onde (x, z) é uma solução ótima do PL.

Apresentadas essas noções iniciais segue o algoritmo de arredondamento probabilístico para o Problema 5.1.

Algoritmo 19: MaxSat-PL$(V, \mathcal{C})$

Entrada: Conjunto de variáveis V e coleção de cláusulas $\mathcal{C}$
Saída: Valoração de variáveis x

```
1 Seja (x', z') uma solução ótima do PL da Satisfatibilidade Máxima
2 para cada v ∈ V faça
3 |   x_v = RAND(x'_v)
4 fim
5 retorna x
```

Veja que o processo realizado pelo algoritmo acima consiste em fazer o arredondamento probabilístico para decidir qual valor atribuir a cada variável v, sendo que o parâmetro da função RAND usada para cada v vem da solução ótima do PL do problema.

Esse algoritmo pode ser considerado eficiente já que sabemos que existem resolvedores de PL eficientes, e as outras partes do algoritmo estão em um laço com tamanho $|V|$, fazendo com que o número de chamadas para RAND também seja polinomial. Dessa forma, o Algoritmo 19 é um algoritmo probabilístico polinomial.

Segue o teorema que enuncia o fator de aproximação desse algoritmo.

Teorema 5.4. *Dada a instância $\langle V, \mathcal{C} \rangle$ para o problema da Satisfatibilidade Máxima, sejam k o número máximo de variáveis em qualquer cláusula em $\mathcal{C}$ e $X_{\mathcal{C}}$ a variável aleatória cujo valor é o número de cláusulas satisfeitas por uma valoração devolvida pelo Algoritmo 19. Então*

$$E[X_{\mathcal{C}}] \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) OPT_{MS}(V, \mathcal{C}).$$

Demonstração. Para cada cláusula $C \in \mathcal{C}$, defina Z_C, variável aleatória, da seguinte forma: se a valoração x, saída do algoritmo, satisfaz C, então $Z_C = 1$, e se a valoração x não satisfaz C, então $Z_C = 0$.

Para que $Z_C = 0$, todas as variáveis v da cláusula C devem ser valoradas de forma que a cláusula C não seja satisfeita. A probabilidade de $[Z_C = 0]$, portanto é dada pelo produto das probabilidades de cada variável não assumir o valor que satisfaz a cláusula, ou seja, x'_v, para $v \in C_0$, e $(1-x'_v)$, para $v \in C_1$.

Se t é a quantidade de variáveis de C, então

$$Pr[Z_C = 1] = 1 - Pr[Z_C = 0] = 1 - \prod_{v \in C_0} x'_v \prod_{v \in C_1} (1 - x'_v).$$

Como a média aritmética de um conjunto de valores é sempre maior ou igual à média geométrica do mesmo conjunto de valores, então

$$\left(\prod_{v\in C_0} x'_v \prod_{v\in C_1}(1-x'_v)\right)^{1/t} \leq \left(\sum_{v\in C_0} x'_v + \sum_{v\in C_1}(1-x'_v)\right)\frac{1}{t}.$$

Dessa forma,

$$Pr[Z_C = 0] = \prod_{v\in C_0} x'_v \prod_{v\in C_1}(1-x'_v) \leq \left(\frac{\sum_{v\in C_0} x'_v + \sum_{v\in C_1}(1-x'_v)}{t}\right)^t.$$

Mas perceba que

$$\sum_{v\in C_0} x'_v + \sum_{v\in C_1}(1-x'_v) = t - \sum_{v\in C_0}(1-x'_v) - \sum_{v\in C_1} x'_v$$

e, pelas restrições do PL da Satisfatibilidade Máxima, sabemos que

$$S := z'_C \leq \sum_{v\in C_0}(1-x'_v) + \sum_{v\in C_1} x'_v.$$

Logo, chegamos ao seguinte resultado:

$$Pr[Z_C = 0] \leq \left(\frac{t-S}{t}\right)^t \leq \left(\frac{t-z'_C}{t}\right)^t = \left(1-\frac{z'_C}{t}\right)^t.$$

Com o resultado obtido acima chegamos na seguinte relação

$$Pr[Z_C = 1] = 1 - Pr[Z_C = 0] \geq 1 - \left(1-\frac{z'_C}{t}\right)^t.$$

Agora, definida a seguinte função $f(z) = 1-(1-\frac{z}{t})^t$, veja que no intervalo $[0,1]$, $f'(z) \geq 0$ e $f''(z) \leq 0$, logo $f(z)$ é concava e crescente no intervalo, sendo que devido a isso $f(z)$ é delimitada inferiormente pela reta entre os pontos $(0, f(0))$ e $(1, f(1))$. Tal reta é dada por $y = f(0)+zf(1)-zf(0)$, mas como $f(0) = 0$, então $f(z) \geq zf(1)$. Esse resultado nos leva até a seguinte

desigualdade:

$$\left(1 - \frac{z'_C}{t}\right)^t \geq \left(1 - \left(1 - \frac{1}{t}\right)^t\right) z'_C.$$

Agora, como $1 - (1/t)^t$ é crescente quando $t \geq 1$ e por hipótese sabemos que $t \leq k$, então

$$Pr[Z_C = 1] \geq \left(1 - \left(1 - \frac{1}{t}\right)\right)^t z'_C \geq \left(1 - \left(1 - \frac{1}{k}\right)\right)^k z'_C.$$

Depois de tudo isso podemos calcular o valor esperado de $X_{\mathcal{C}}$:

$$E[X_{\mathcal{C}}] = \sum_{C \in \mathcal{C}} E[Z_C] = \sum_{C \in \mathcal{C}} Pr[Z_C = 1].$$

Devido às desigualdades obtidas, podemos concluir que

$$E[X_{\mathcal{C}}] \geq \sum_{C \in \mathcal{C}} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) z'_C \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) OPT_{MS}(V, \mathcal{C}),$$

sendo que a última desigualdade segue do fato de que z'_C é solução ótima do PL e, como sabemos, isso significa que é maior ou igual do que a solução ótima do problema original. Com isso provamos o teorema. □

O teorema a seguir segue diretamente do resultado do Teorema 5.4 e do fato de que

$$\lim_{k \to \infty} \left(1 - \frac{1}{k}\right)^k = \frac{1}{e} < 0,37.$$

Teorema 5.5. *O Algoritmo 19 é uma* $0,63$*-aproximação probabilística polinomial para o problema da Satisfatibilidade Máxima.*

O primeiro algoritmo apresentado para a Satisfatibilidade Máxima (Algoritmo 18) costuma devolver um resultado melhor quando as cláusulas têm muitas variáveis. Já o segundo algoritmo (Algoritmo 19) obtém melhores resultados quando as cláusulas possuem no máximo duas variáveis [2]. Devido

à característica dos algoritmos probabilísticos serem eficientes, parece natural tentar combinar os dois algoritmos até agora estudados e obter um um novo algoritmo. Será justamente essa combinação o último algoritmo para a Satisfatibilidade Máxima que apresentaremos. Segue o algoritmo.

Algoritmo 20: MaxSat-Combinado($V, \mathcal{C}$)

Entrada: Conjunto de variáveis V e coleção de cláusulas $\mathcal{C}$
Saída: Valoração de variáveis x

```
x_1 = MaxSat-Johnson(V)
x_2 = MaxSat-PL(V,C)
Seja n_1 o número de cláusulas satisfeitas por x_1
Seja n_2 o número de cláusulas satisfeitas por x_2
se n_1 ≥ n_2 então
    x = x_1
fim
senão
    x = x_2
fim
retorna x
```

O interessante do algoritmo acima é que ele apresenta um fator de aproximação probabilístico melhor do que os dos outros dois algoritmos. Segue o teorema que enuncia tal fator de aproximação.

Teorema 5.6. *Dada uma instância $\langle V, \mathcal{C} \rangle$ para a Satisfatibilidade Máxima, o Algoritmo 20 é uma 0,75-aproximação probabilística polinomial da Satisfatibilidade Máxima.*

Demonstração. Vamos considerar que C_i é o conjunto de todas as cláusulas com i variáveis

Sejam X_C, X_{C1} e X_{C2} variáveis aleatórias que indicam quantas cláusulas são satisfeitas por uma valoração produzida pelos Algoritmos 20, 18 e 19, respectivamente.

Veja que X_C pode assumir o valor de X_{C1} ou de X_{C2}. Se $X_C = X_{C1}$, então

$$X_C = X_{C1} \geq X_{C2} \Rightarrow X_C \geq \frac{1}{2}(X_{C1} + X_{C2}),$$

e o caso em que $X_C = X_{C2}$ apresenta um resultado análogo. Esse resultado em conjunto com a linearidade da esperança de variáveis aleatórias nos leva até a seguinte desigualdade:

$$E[X_C] \geq E\left[\frac{1}{2}(X_{C1} + X_{C2})\right] = \frac{1}{2}E\left[X_{C1} + X_{C2}\right] = \frac{1}{2}E\left[X_{C1}\right] + E\left[X_{C2}\right].$$

Antes de prosseguirmos vamos definir que:

- k é o número máximo de variáveis em qualquer cláusula em $\mathcal{C}$; e
- z'_c é a solução ótima do PL do MaxSat com instância $\langle V, \mathcal{C} \rangle$.

Da demonstração do Teorema 5.2, que demonstra o fator de aproximação do MaxSat-Johnson, temos que

$$E[X_{C1}] \geq |\mathcal{C}| \left(1 - \frac{1}{2^k}\right) \geq \sum_k \sum_{c \in C_k} \left(1 - \frac{1}{2^k}\right) z'_c.$$

Já da demonstração do Teorema 5.4, do fator de aproximação do MaxSat-PL, temos que

$$E[X_{C2}] \geq \sum_k \sum_{c \in C_k} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) z'_c.$$

Veja que temos dois somatórios em cada uma das expressões acima como uma forma de garantir que todas as cláusulas sejam consideradas. Juntando as duas relações apresentadas, obtemos que

$$E[X_C] \geq \frac{1}{2} \sum_k \sum_{c \in C_k} \left[\left(1 - \frac{1}{2^k}\right) + \left(1 - \left(1 - \frac{1}{k}\right)^k\right)\right] z'_c.$$

Vamos agora dividir nossa análise em três casos, cada um analisando um conjunto de cláusulas com valores de k diferentes. Nossa análise consistirá em entender o quanto cada um desses tipos de cláusulas contribui para o somatório dado acima.

Vamos começar pelo caso em que temos uma cláusula em C_1, ou seja, $k = 1$. Tal cláusula contribuiria para o somatório acima com o valor

$$1 - \frac{1}{2} + 1 - (1 - 1) = \frac{3}{2}.$$

Agora no segundo caso considere uma cláusula C_2, ou seja, $k = 2$. Esse tipo de cláusula contribui com

$$1 - \frac{1}{2^2} + 1 - \left(1 - \frac{1}{2}\right)^2 = \frac{3}{2}.$$

Para o terceiro e último tipo de cláusula considere $C_{\geq 3}$, ou seja, $k \geq 3$. Para esse tipo de cláusula vamos definir uma função

$$f(w) = \left(\left(1 - \frac{1}{2^w}\right) + \left(1 - \left(1 - \frac{1}{w}\right)^w\right)\right).$$

Para $w \geq 2$ essa função é estritamente crescente, além disso também veja que

$$\lim_{w \to \infty} f(w) = 2 - \frac{1}{e} \geq \frac{3}{2}.$$

Assim, podemos concluir que para $k \geq 3$ a cláusula sempre contribui com valor $\geq \frac{3}{2}$ para o somatório.

Antes de continuarmos vamos definir que

$$S := \frac{1}{2} \sum_k \sum_{c \in C_k} \left[\left(1 - \frac{1}{2^k}\right) + \left(1 - \left(1 - \frac{1}{k}\right)^k\right)\right] z'_c.$$

Agora, com os três possíveis valores de k analisados podemos concluir que

$$E[X_C] \geq S \geq \frac{1}{2} \sum_k \sum_{c \in C_k} \frac{3}{2} z'_c,$$

sendo que isso nos leva a

$$E[X_C] \geq \frac{3}{4} \sum_k \sum_{c \in C_k} z'_c,$$

mas veja que agora esse somatório representa o valor da solução do PL da satisfatibilidade máxima, valor com certeza maior do que o da solução ótima para o problema combinatório. Então

$$E[X_C] \geq \frac{3}{4} OPT_{MS}(V, \mathcal{C}).$$

□

Capítulo 6

Cobertura por Conjuntos por Programação Linear

Nesse capítulo mostraremos o uso da programação linear (PL) em problemas computacionais. Para isso, apresentaremos como a PL pode ser usada para analisar e projetar algoritmos para um problema. O problema que analisaremos nesse capítulo será o da Cobertura por Conjuntos, já apresentado no Capítulo 2 (Problema 2.10).

Começaremos o capítulo apresentando uma técnica de análise de algoritmos chamada *dual-fitting*, que usa a dualidade de PLs. Depois, apresentaremos algoritmos de aproximação para a Cobertura por Conjuntos que usam a técnica de arredondamento e a técnica de primal-dual.

6.1 Cobertura por Conjuntos e *Dual-Fitting*

O ***dual-fitting*** é um técnica de análise usada em algoritmos combinatórios que usa a dualidade dos programas lineares para estabelecer qual o fator de aproximação do algoritmo. Nessa seção analisaremos um algoritmo combinatório para a Cobertura por Conjuntos através do *dual-fitting*, provando que tal algoritmo é uma H_n-aproximação do problema, como o Algoritmo 7,

apresentado para o mesmo problema na Seção 2.5.

Para entender como o uso do *dual-fitting* nos dá um fator de aproximação para um algoritmo, temos que primeiro entender como um algoritmo para um problema pode ser relacionado ao dual de sua relaxação linear. Para facilitar, vamos explicar essas coisas através de um exemplo, que no caso é o problema da Cobertura por Conjuntos. Para começar, vamos descrever o problema da Cobertura por Conjuntos (Problema 2.10) através de um PLI de minimização.

Problema 6.1 (PLI da Cobertura por Conjuntos)**.** *Dados um conjunto finito E, uma coleção de conjuntos $\mathcal{S}$, composta por subconjuntos de E, e uma função de custo $c : \mathcal{S} \to \mathbb{Q}^+$. Associe a cada $S \in \mathcal{S}$ uma variável binária (só pode assumir valor 0 ou 1) x_S. Nós desejamos minimizar a função*

$$\sum_{S \in \mathcal{S}} c(S) x_S,$$

sendo que as seguintes restrições são impostas

$$\forall e \in E, \sum_{S : e \in S} x_S \geq 1.$$

Veja que da forma que a Cobertura por Conjuntos está definida acima, nós desejamos minimizar o custo dos conjuntos escolhidos, sendo que cada conjunto é representado por uma variável binária em nossa função objetivo. Além disso, veja as restrições, nada mais são do que uma forma de dizer que dentre todos os conjuntos que contém o elemento $e \in E$, ao menos um desses conjunto deve ser escolhido.

Feito o PLI do problema partimos em busca de sua relaxação linear. Para o caso da Cobertura por Conjuntos, sua relaxação linear teria o seguinte significado: podemos pegar certas porcentagens de cada conjunto S para compor uma solução final. Assim, segue a relaxação linear do problema.

Problema 6.2 (Primal da Cobertura por Conjuntos)**.** *Dados um conjunto finito E, uma coleção de conjuntos $\mathcal{S}$, composta por subconjuntos de E, e uma função de custo $c : \mathcal{S} \to \mathbb{Q}^+$. Associe a cada $S \in \mathcal{S}$ uma variável $x_S \in [0, 1]$. Nós desejamos minimizar a função*

$$\sum_{S \in \mathcal{S}} c(S) x_S,$$

sendo que as seguintes restrições são impostas:

$$\forall e \in E, \sum_{S : e \in S} x_S \geq 1.$$

Pode parecer estranho que as restrições ainda exijam que a soma das variáveis x_S, para os conjuntos S que contém um determinado elemento e, seja ao menos 1. A questão é que não temos certeza quais elementos dos conjuntos estaremos selecionando, já que agora, no PL, podemos tomar uma parte fracionária de um conjunto para compor a cobertura devolvida. Assim, a única maneira de termos certeza que todos os elementos de E estão sendo cobertos é através dessa restrição.

Agora, uma parte menos trivial que as outras, vamos desenvolver o dual da Relaxação de Cobertura por Conjuntos. Sabemos que para desenvolver o dual nós temos que introduzir novas variáveis relacionadas a uma combinação linear das restrições. Segue o dual do Problema 6.2.

Problema 6.3 (Dual da Cobertura por Conjuntos)**.** *Dados um conjunto finito E, uma coleção de conjuntos $\mathcal{S}$, composta por subconjuntos de E, e uma função de custo $c : \mathcal{S} \to \mathbb{Q}^+$. Associe a cada elemento $e \in E$ uma variável $y_e \geq 0$. Nós desejamos maximizar a função*

$$\sum_{e \in E} y_e,$$

sendo que as seguintes restrições são impostas:

$$\forall S \in \mathcal{S}, \sum_{e \in S} y_e \leq c(S).$$

Não vamos tentar encontrar um significado para o dual do problema pois, como diz Vazirani [6], mais importante do que entendermos o significado intuitivo do dual é entendermos que trata-se de um processo mecânico que sempre nos resulta em um novo programa linear que está diretamente relacionado com o primal.

Agora, segue o algoritmo combinatório que analisaremos nessa seção através do *dual-fitting*. Esse é um algoritmo guloso que se baseia em uma relação custo/cobertura para tomar suas decisões.

Algoritmo 21: COBERTURA-PL(E, $\mathcal{S}$, c)

Entrada: Conjunto E, coleção de conjuntos finitos $\mathcal{S}$ e função de custo nos conjuntos de $\mathcal{S}$
Saída: Cobertura T de E

```
1 C = ∅
2 T = ∅
3 enquanto C ≠ E faça
4 |   Encontre o conjunto S ∈ 𝒮 tal que α = c(S)/|S − C| é mínimo
5 |   Para cada e ∈ S − C defina p(e) = α, caso S seja o primeiro
  |    conjunto selecionado que cobre e
6 |   C = C ∪ S
7 |   T = T ∪ {S}
8 fim
9 retorna T
```

Sendo $OPT_{PD}(E, \mathcal{S}, c)$ o custo da solução ótima dos problemas primal e dual, $OPT_{CC}(E, \mathcal{S}, c)$ o custo da solução ótima da Cobertura por Conjuntos e $c(T)$ o custo da solução devolvida pelo nosso algoritmo, veja que, independente da solução viável devolvida pelo algoritmo, é verdade que

$$OPT_{PD}(E, \mathcal{S}, c) \leq OPT_{CC}(E, \mathcal{S}, c) \leq c(T). \tag{6.1}$$

Também não podemos nos esquecer que o custo de qualquer solução viável do dual tem custo menor que o custo de uma solução ótima para o problema original. Vamos analisar a relação dos problemas primal e dual com nosso algoritmo. É importante observar que mesmo que o primal e o dual possuam uma relação matemática entre si, eles podem ser tratados de maneira independente.

Vamos começar analisando a relação do algoritmo com o primal, algo mais fácil de ser visualizado. É evidente, por construção, que qualquer cobertura T devolvida pelo nosso algoritmo representa uma atribuição das variáveis x_S. Além disso, qualquer cobertura T devolvida é uma solução viável do problema original da Cobertura por Conjuntos, logo também é solução viável de seu PLI e de sua relaxação linear. Concluindo, o nosso algoritmo sempre consegue encontrar uma atribuição de variáveis válida para o primal.

Agora, vamos refletir um pouco sobre as variáveis y_e do dual. A questão que surge é: como podemos usar nosso algoritmo combinatório para encontrar uma atribuição das variáveis y_e? Lembre-se que por mais que a relação primal-dual exista, ambos não são o mesmo problema. Pode parecer que encontrar uma forma de atribuir valores aos y_e é simples, já que qualquer solução viável do primal pode ser traduzida para uma solução do dual. De fato, qualquer solução viável do primal pode ser traduzida para uma solução do dual, mas nem sempre essa solução obtida para o dual é viável. É justamente nessa atribuição de valores às variáveis do dual que encontra-se o coração do *dual-fitting*.

Observe que a atribuição de variáveis realizadas no primal está relacionada aos conjuntos, enquanto a atribuição de variáveis do dual está relacionada aos elementos. Veja que na linha 4 nós definimos uma função p que atribui o valor α (custo/cobertura) para cada elemento e, sendo que o valor

α depende do conjunto selecionado para cobrir e. Se estabelecermos que

$$y_e = p(e),$$

temos uma atribuição de valores para as variáveis y_e que relaciona o nosso algoritmo com uma possível solução do dual. Porém, devemos verificar se essa forma de atribuir valores sempre dá uma solução viável para o dual. No caso dessa atribuição a resposta é não. Veja no exemplo a seguir.

Exemplo 6.1. *Sejam* $S_1 = \{1, 2, 3\}, S_2 = \{3, 5, 7\}, S_3 = \{2, 4, 5\}, S_4 = \{3, 4, 5\}, S_5 = \{6, 7, 8\}$*, sendo que* $\forall S_i$ *vale que* $c(S_i) = |S_i|$*, desejamos encontrar uma cobertura para o conjunto* $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$*.*

Considerando que a solução devolvida pelo Algoritmo 21 seja $T = \{S_1, S_4, S_5\}$*, então obtemos uma solução viável para o problema, seu PLI e sua relaxação linear.*

Mas veja que ao considerarmos $y_e = p(e)$*, então obtemos, para* S_4*,*

$$\sum_{e \in S_4} y_e = p(3) + p(4) + p(5) = 1 + 1,5 + 1,5 \geq 3 = c(S_4),$$

ou seja, desrespeitamos a restrição do dual, logo não temos uma solução viável para o dual.

Nesse ponto, duas dúvidas podem surgir. A primeira seria sobre como encontrar uma atribuição para y_e, usando nosso algoritmo, que sempre seja viável. Essa é uma pergunta válida que é respondida com o lema a seguir.

Lema 6.1. *Dados uma instância* $\langle \mathcal{S}, E, c \rangle$ *para a Cobertura por Conjuntos e o Algoritmo 21 para esse problema. Se considerarmos a seguinte atribuição de valores para as variáveis* y_e *do Dual da Cobertura por Conjuntos sempre obteremos uma resposta viável para esse dual:*

$$y_e = \frac{p(e)}{H_n}.$$

Demonstração. Para que o lema seja válido temos que demonstrar que a atribuição de valores para y_e sempre respeita as restrições do problema.

Considere um conjunto $S \in \mathcal{S}$ tal que $|S| = k$. Vamos enumerar todos os elementos de S pela ordem que eles são cobertos pelo algoritmo, sendo que empates podem ser decididos de maneira arbitrária. Assim, obtemos $e_1, e_2, \ldots, e_k$.

Considere a iteração que o algoritmo cobre o elemento e_i. Veja que nessa iteração, antes de cobrir o elemento, temos ao menos $k-i+1$ elementos ainda não cobertos. Então, nessa iteração, o custo da cobertura de S é no máximo $\frac{c(S)}{k-i+1}$. Como sabemos que o algoritmo escolhe para cobrir e_i o conjunto com melhor custo-cobertura vale que

$$y_{e_i} \leq \frac{1}{H_n} \frac{c(S)}{k-i-1}.$$

Sabendo que a desigualdade acima vale para toda iteração que cobre um elemento de S, obtemos que

$$\sum_{i=1}^{k} y_{e_i} \leq \frac{c(S)}{H_n} \sum_{i=1}^{k} \frac{1}{i} = \frac{H_k}{H_n} c(S) \leq c(S).$$

Com isso, concluímos que as condições de restrição do dual são respeitadas para todo conjunto $S \in \mathcal{S}$, e obtemos uma forma de atribuir valores a y_e que usa o algoritmo e que sempre devolve uma solução viável para o dual.

□

Agora, a segunda dúvida que surge é como tal atribuição para y_e nos ajudaria na demonstração do fator de aproximação do algoritmo. Isso é respondido no seguinte teorema.

Teorema 6.1. *O Algoritmo 21 é uma H_n-aproximação para o problema da Cobertura por Conjuntos.*

Demonstração. Seja $I = \langle \mathcal{S}, E, c \rangle$ uma instância da Cobertura por Conjuntos.

Sejam $e_1, \ldots, e_k \in E$ elementos que sejam cobertos pela primeira vez no algoritmo pelo conjunto $S \in \mathcal{S}$. Logo,

$$c(S) = \sum_{i=1}^{k} p(e_i).$$

Veja que a relação acima é verdade para todo conjunto que for selecionado para compor a cobertura T devolvida. Logo, vale que

$$c(T) = \sum_{e \in E} p(e) = H_n \sum_{e \in E} y_e \leq H_n OPT_{PD}(I) \leq H_n OPT_{CC}(I)$$

sendo que as duas últimas desigualdades valem devido à equação (6.1). □

Agora que conseguimos provar o fator de aproximação do algoritmo usando o dual do programa, vamos fazer um resumo da técnica de *dual-fitting* através de uma lista de passos que a técnica usa. Para isso, considere um problema de otimização combinatória e considere que nós temos um algoritmo combinatório que devolve soluções viáveis do problema. Segue a lista:

1. Encontre o PLI do problema;
2. Encontre a relaxação linear do problema e considere ela o primal;
3. Encontre o dual da relaxação linear;
4. Encontre uma função para atribuir valores às variáveis do dual e que tenha relação de custo com as soluções devolvidas pelo algoritmo;
5. Verifique se a função encontrada sempre dá uma atribuição que resulte em solução viável;
6. Baseado na relação entre a função e o valor de uma solução viável para o problema original encontre o fator de aproximação.

Não se esqueça o coração da técnica está na forma como atribuímos valores às variáveis do dual. Também, é bom citar que esse fator de aproximação é justo para o Algoritmo 21, como mostra o exemplo dado ao final da Seção 2.5.

6.2 Cobertura por Conjuntos por Arredondamento

Nessa seção apresentaremos dois algoritmos de aproximação para a Cobertura por Conjuntos (Problema 2.10) que usam a técnica de arredondamento. O primeiro algoritmo que apresentaremos pode ser chamado de algoritmo de arredondamento simples, sendo que seu fator de aproximação está relacionado à solução viável devolvida. O segundo algoritmo já não pode ser considerado simples, pois faz uso de aleatorização no arredondamento.

Vamos começar pelo algoritmo que usa arredondamento simples. Na seção anterior já apresentamos o PLI da Cobertura por Conjuntos (Problema 6.1) e sua relaxação linear (Problema 6.2). Uma maneira simples de arredondar uma solução do PL para uma solução do PLI é alterando o valor de todas as variáveis x do PL que têm valor maior ou igual a 0 para 1, ou seja, todo conjunto que foi parcialmente tomado no PL é tomado inteiro no PLI. Além de considerar a ideia de arredondamento comentada, vamos adicionar uma heurística também muito simples que geralmente diminui a quantidade de conjuntos selecionados. O fator de aproximação desse algoritmo é f, onde f é a quantidade de conjuntos máxima que qualquer elemento $e \in E$ participa, ou seja, em quantos conjuntos o elemento mais frequente aparece. Segue o algoritmo.

Veja que no Algoritmo 22 temos um procedimento muito simples que usa uma solução ótima da relaxação linear da Cobertura por Conjuntos para nos devolver uma solução viável para o problema. Na linha 2 obtemos uma solução ótima do PL da Cobertura por Conjuntos, o conjunto U possui uma atribuição de variáveis que representa uma resposta ótima desse PL. Na

Algoritmo 22: COBERTURA-ARREDONDAMENTOSIMPLES(E, $\mathcal{S}$, c)

Entrada: Conjunto E, coleção de conjuntos finitos $\mathcal{S}$ e função de custo nos conjuntos de $\mathcal{S}$
Saída: Cobertura T de E

1 $T = \emptyset$
2 Seja x a solução do PL para a instância $\langle E, \mathcal{S}, c \rangle$
3 Adicione a T todo conjunto $S \in \mathcal{S}$ tal que $x_s \geq 1/f$
4 **retorna** T

linha 3 construímos a solução viável T para a Cobertura por Conjuntos com base no critério citado.

Agora, vamos demonstrar que esse algoritmo extremamente simples é uma f-aproximação para o problema no seguinte teorema.

Teorema 6.2. *Dada uma instância $I = \langle E, \mathcal{S}, c \rangle$ da Cobertura por Conjuntos, o Algoritmo 22 é uma f-aproximação para o problema.*

Demonstração. Dada a coleção T de conjuntos escolhidos pelo algoritmo e um elemento arbitrário $e \in E$, sabemos que e está presente em no máximo f conjuntos.

Pela condição de restrição do PL da Cobertura por Conjuntos (Problema 6.2), sabemos que a soma dos x_S para todos os conjuntos que contêm o elemento e deve ser ao menos 1. Como e faz parte de no máximo f conjuntos, ao menos um desses conjuntos possui $x_S \geq 1/f$, caso contrário não haveria como atender a restrição. A junção dessas coisas garante que o algoritmo sempre devolve uma solução viável para o problema.

O arredondamento feito para obter uma solução viável para a Cobertura por Conjuntos aumenta o custo de cada conjunto já selecionado por um fator máximo de f, logo $c(T) \leq f.OPT_{Primal}(I)$. Lembrando que x é a solução ótima da relaxação e T é a solução viável devolvida pelo algoritmo.

Assim, podemos concluir que

$$OPT_{Primal}(I) \leq OPT_{CC}(I) \leq c(T) \Rightarrow c(T) \leq f \cdot OPT_{Primal}(I) \leq f.OPT_{CC}(I)$$

□

O Algoritmo 22 é extremamente simples e como temos métodos de resolução de PLs em tempo polinomial o algoritmo acaba sendo polinomial, devolvendo uma solução não só viável mas com garantia de aproximação.

Apresentaremos agora um segundo algoritmo para Cobertura por Conjuntos que usa a técnica de arredondamento do PL. Esse algoritmo é probabilístico e se baseia na ideia de arredondamento probabilístico, como o Algoritmo 18 no Capítulo 5. A ideia do algoritmo é pegar uma solução ótima para o PL e realizar sorteios enviesados usando os valores fracionários atribuídos pela solução ótima do PL, que determinam, para cada conjunto, se ele faz parte da solução. Diferente de outros algoritmos apresentados, desenvolveremos primeiro a ideia do algoritmo matematicamente e só depois apresentaremos o algoritmo em si, pois esse algoritmo depende de alguns conceitos que são melhor introduzidos em sua prova.

Então vamos começar a apresentar a notação que usaremos pelo resto da seção. Vamos considerar que tenhamos uma instância $\langle E, \mathcal{S}, c\rangle$ para a Cobertura por Conjuntos e uma solução ótima x para a relaxação linear desse problema, ou seja, para cada $S \in \mathcal{S}$ temos um elemento x_S associado referente à solução x. Vamos agora definir uma função probabilidade $p(S)$ para cada $S \in \mathcal{S}$, sendo que esse valor representa a probabilidade desse conjunto S compor a solução para o nosso problema. Para determinarmos se cada S compõem a solução vamos usar um RAND (Gerador aleatório de bits), sendo que $p(S)$ dependerá desse RAND. Se RAND devolver 1, então S participa da solução, mas caso RAND devolva 0, então S não participa da solução. O processo de sorteio onde temos uma coleção de conjuntos como resposta não é feito só uma vez, mas sim é realizado diversas vezes por razões probabilísticas, cujos detalhes serão explicados mais a frente.

Seja $\mathcal{C}$ a coleção dos conjuntos $S \in \mathcal{S}$ tais que $\text{RAND}(p(S)) = 1$, ou seja, os conjuntos devolvidos em uma iteração do algoritmo. O valor esperado do

custo de $\mathcal{C}$ é determinado por

$$E[c(\mathcal{C})] = \sum_{S \in \mathcal{S}} p(S).c(S) = \sum_{S \in \mathcal{S}} x_S.c(S) = OPT_{PL}(E, \mathcal{S}, c),$$

sendo que o último termo é o custo da solução ótima da relaxação linear do problema. Além disso, lembramos que $n = |E|$.

Agora, vamos falar sobre a probabilidade de um elemento $a \in E$ ser coberto por $\mathcal{C}$, a coleção de conjuntos devolvida ao final de um sorteio. Lembrando que estamos considerando que os conjuntos S não possuem elementos fora de E. Suponha que a pertença a k conjuntos em $\mathcal{S}$. Sabemos que cada um desses k conjuntos possui uma probabilidade $p(S)$ associada. Vamos então indexar esses conjuntos e essas probabilidades, o que nos dá os conjuntos $S_1, \ldots, S_k$ e as probabilidades associadas $p(S_1), \ldots, p(S_k)$.

Lembrando das restrições do PL da Cobertura por Conjuntos sabemos que para o nosso elemento a é verdade que $p(S_1) + \cdots + p(S_k) \geq 1$. Dessa forma, é possível ver que a probabilidade de a ser coberto é minimizada, de forma a ainda atender as restrições, quando todo $p(S_i) = 1/k$. Então, a probabilidade de a não ser coberto nessa situação é dada pelo produto da probabilidade dos k conjuntos não serem selecionados, ou seja,

$$Pr[a \text{ não é coberto por } \mathcal{C}] \leq \left(1 - \frac{1}{k}\right)^k,$$

o que implica que a probabilidade dele ser coberto é dada por

$$Pr[a \text{ é coberto por } \mathcal{C}] \geq 1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e},$$

sendo que os sinais de desigualdade são usados pois estamos assumindo valores de probabilidade que minimizem ao máximo a probabilidade de a ser coberto, não se trata de uma atribuição de probabilidades única.

Agora, consideremos uma constante d que respeite a seguinte desigual-

dade

$$\left(\frac{1}{e}\right)^{d\ln(n)} \leq \frac{1}{4n},$$

onde $d\ln(n)$ é a quantidade de vezes que repetiremos o processo de sorteio. Então, a cada uma dessas vezes teremos como resposta uma subcoleção dos conjuntos S, sendo que a resposta final do algoritmo é a união de todas essas subcoleções, que chamaremos de $\mathcal{C}'$. Como repetimos o processo diversas vezes, vamos indexar as coleções de conjuntos devolvidas a cada iteração como $\mathcal{C}_1, \ldots, \mathcal{C}_{d\ln(n)}$.

Considerando que repetiremos o processo $d\ln(n)$ vezes, veja que podemos determinar a probabilidade do nosso elemento a não ser coberto ao final de todas essas iterações, tal valor é dado por

$$Pr[a \text{ não é coberto por } \mathcal{C}'] \leq \left(\frac{1}{e}\right)^{d\ln(n)} \leq \frac{1}{4n}.$$

Podemos ver pela desigualdade acima que a escolha do valor de d não foi arbitrária.

Agora vamos falar sobre a probabilidade de $\mathcal{C}'$ não ser uma cobertura viável para o problema. Para que isso aconteça, algum dos elementos pertencentes a E não foi coberto. A probabilidade de tal evento é dada pela soma das probabilidades de um dos n elementos não ser coberto, ou seja,

$$Pr[\mathcal{C}' \text{ não ser cobertura}] \leq n\frac{1}{4n} = \frac{1}{4}.$$

Devido à linearidade das esperanças e devido à quantidade de coleções geradas ser $d\ln(n)$, o seguinte resultado é válido

$$E[c(\mathcal{C}')] = \sum_{i=1}^{d\ln(n)} E[c(\mathcal{C}_i)] \leq d\ln(n)OPT_{PL}(E, \mathcal{S}, c).$$

Mas veja que como $OPT_{PL}(I) \leq OPT_{CC}(I)$ para toda instância, o re-

sultado acima prova que o algoritmo descrito é uma $O(\ln(n))$-aproximação probabilística para o problema da Cobertura por Conjuntos. Isso em si é um resultado significativo, porém, podemos ir mais longe ao falarmos sobre esse algoritmo.

Aplicando a Desigualdade de Markov (Teorema 5.1), teorema visto no Capítulo 5, e considerando que $d\ln(n)OPT_{PL}(E, \mathcal{S}, c) = t$, obtemos que

$$Pr[c(\mathcal{C}') \geq 4t] \leq \frac{E[c(\mathcal{C}')]}{4t} \leq \frac{t}{4t} = \frac{1}{4}.$$

Mas afinal para que serve essa última constatação? Porque veja que nós obtivemos limitantes para o valor da probabilidade dos eventos não desejados, que são

- $\mathcal{C}'$ não ser uma cobertura, onde vimos que a probabilidade de tal evento é $\leq 1/4$;
- O custo da cobertura $\mathcal{C}'$ devolvida ser superior a $OPT_{CC}(E, \mathcal{S}, c)$ por um fator maior que $d\ln(n) = O(\ln(n))$.

Dessa forma, obtivemos dois resultados relevantes que referem-se à qualidade da solução devolvida. Também podemos concluir que:

$$Pr[\mathcal{C}' \text{ não ser cobertura válida ou } c(\mathcal{C}') \geq 4t] \leq \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$$

logo,

$$Pr[\mathcal{C}' \text{ ser cobertura válida e } c(\mathcal{C}') \leq 4t] \geq \frac{1}{2}.$$

Veja que por mais que não tenhamos formalizado um teorema para esse algoritmo, ainda assim conseguimos definir seu fator de aproximação probabilístico e descobrir alguns outros detalhes sobre o mesmo, sempre justificando os resultados.

Para finalizar a seção, segue uma formalização do algoritmo descrito, que não termina a execução até encontrar uma solução aceitável para o problema.

Algoritmo 23: COBERTURA-RANDOMIZADO(E, $\mathcal{S}$, c)

Entrada: Conjunto E, coleção de conjuntos finitos $\mathcal{S}$ e função c de custo dos conjuntos de $\mathcal{S}$
Saída: Cobertura $\mathcal{C}'$ de E

Seja x a solução do programa linear do problema
enquanto $\mathcal{C}'$ *não é cobertura de* E *ou* $c(\mathcal{C}') \geq 4tc(x)$ **faça**
    para $i = 1$ *até* $d\ln(n)$ **faça**
        Adicione cada $S \in \mathcal{S}$ a $\mathcal{C}_i$ com probabilidade $p(S)$ e de maneira independente usando RAND($p(S)$)
    fim
    $\mathcal{C}' = \bigcup_{i \leq d\ln(n)} \mathcal{C}_i$
fim
retorna $\mathcal{C}'$

6.3 Cobertura por Conjuntos por Esquema Primal-Dual

Como já discutimos no Capítulo 4, o esquema primal-dual nos dá um algoritmo combinatório com um bom fator de aproximação e um bom tempo de execução [6], por isso podemos falar que ele é o método preferível para o projeto de algoritmos de aproximação.

Nessa seção, apresentaremos primeiro uma visão geral do esquema primal-dual e depois uma f-aproximação para a Cobertura por Conjuntos, onde f é a quantidade de conjuntos em que o elemento mais frequente aparece.

O esquema primal-dual tem suas origens no projeto de algoritmos exatos sendo responsável por algoritmos extremamente eficientes para alguns problemas em $\mathcal{P}$ [6]. Também usa-se muito as condições de folga complementares para o projeto desse tipo de algoritmo (Teorema 4.4).

Vamos começar com uma visão geral sobre o esquema primal-dual. Para isso, vamos considerar o seguinte problema de minimização genérico como nosso primal e em seguida apresentaremos seu dual.

Problema 6.4 (Primal)**.** *Dada uma entrada que especifique* a_{ij}, b_i *e* c_j *dese-*

jamos minimizar

$$\sum_{j=1}^{n} c_j x_j,$$

sujeito às seguintes condições

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad \forall i \in \{1, \ldots, m\}$$

$$x_j \geq 0 \qquad \forall j \in \{1, \ldots, n\}$$

Problema 6.5 (Dual)**.** *Dada uma entrada que especifique a_{ij}, b_i e c_j desejamos maximizar*

$$\sum_{i=1}^{m} b_i y_i,$$

sujeito às seguintes condições

$$\sum_{i=1}^{m} a_{ij} y_i \leq c_j \qquad i \in \{1, \ldots, m\}$$

$$y_i \geq 0 \qquad j \in \{1, \ldots, n\}$$

Apresentados esses PLs podemos aplicar neles as Condições de Folga Complementares (Teorema 4.4) da seguinte forma.

Condição de folga do primal: Seja $\alpha \geq 1$, então

$$\forall j \in \{1, \ldots, n\} \text{ vale que ou } x_j = 0, \text{ ou } c_j/\alpha \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j.$$

Condição de folga do dual: Seja $\beta \geq 1$, então

$$\forall i \in \{1, \ldots, m\} \text{ vale que ou } y_i = 0, \text{ ou } b_i \leq \sum_{i=1}^{n} a_{ij} x_j \leq \beta b_i.$$

Veja que devido às variáveis α e β, a forma como as condições de folga

foram descritas acima não é igual à forma como está descrito no Teorema 4.4. Eventualmente, se uma dessas variáveis assumir o valor de 1 com certeza uma das condições será satisfeita. A maioria dos algoritmos que usa esquema primal-dual relaxa uma dessas condições mas impõe a outra. Agora, segue uma proposição que usa das condições de folga da forma que foram descritas acima.

Proposição 6.1. *[6] Se x e y são, respectivamente, soluções viáveis do primal e do dual de um PL e que respeitam as condições de folga descritas acima, então*

$$\sum_{j=1}^{n} c_j x_j \leq \alpha\beta \sum_{i=1}^{m} b_i y_i.$$

A proposição acima nos ajudará a demonstrar como o esquema primal-dual é extremamente viável como ferramenta em algoritmos de aproximação.

Sobre os algoritmos primal-dual, eles começam com uma solução inviável para o primal e uma solução viável para o dual, normalmente as soluções triviais $x = 0$ e $y = 0$. Depois desse começo vão sendo feitas iterações que vão incrementando a viabilidade de x e a qualidade de y. Quando falamos viabilidade de x queremos dizer que x pode começar como solução inviável, porém, a cada iteração x vai se aproximando dos valores de uma solução viável do problema.

Ao final de um algoritmo primal-dual, obtemos uma solução viável do primal que respeite as condições de folga, para valores de α e β cabíveis. A solução primal obtida é sempre convertida em uma solução para a versão inteira do problema, e a solução dual é usada como limitante inferior para a solução ótima do problema. Isso em combinação com a Proposição 6.1 nos dá que

$$\sum_{j=1}^{n} c_j x_j \leq \alpha\beta \sum_{i=1}^{m} b_i y_i \leq \alpha\beta OPT_P.$$

Logo, um algoritmo que use o esquema primal-dual é uma $\alpha\beta$-aproximação para o problema. Na desigualdade acima, OPT_P é o custo de uma solução

ótima do problema.

A forma como a construção do primal-dual ocorre se dá por melhoras uma-a-uma, ou seja, a cada iteração o primal sugere para o dual como melhorar o primal, e vice versa [6].

Após essa descrição mais geral sobre o método primal-dual e como algoritmos desse tipo funcionam, vamos voltar nossas atenções ao problema da Cobertura por Conjuntos. Lembrando que as descrições do problema (Problema 2.10), seu PL (Problema 6.2) e o seu dual (Problema 6.3), já foram apresentados.

Vamos aqui apresentar um algoritmo para a Cobertura por Conjuntos que possui fator de aproximação f, onde f é a quantidade de conjuntos em que o elemento mais frequente aparece. Para esse algoritmo vamos considerar que $\alpha = 1$ e $\beta = f$.

Com os valores que definimos para α e β, vamos agora analisar como essa decisão afeta as condições de folga do primal e do dual para a Cobertura por Conjuntos.

Primal: $\forall S \in \mathcal{S}$: Se $x_S \neq 0$ deve valer que

$$c(S)/1 \leq \sum_{e \in S} y_e \leq c(S) \Rightarrow \sum_{e \in S} y_e = c(S).$$

Dual: $\forall e$: Se $y_e \neq 0$ deve valer que

$$1 \leq \sum_{S \in \mathcal{S}} x_S \leq f.$$

Referente à condição de folga do primal descrita acima, vamos definir que um conjunto S do primal é justo se $\sum_{e \in S} y_e = c(S)$, ou seja, a soma de todos os y_e referentes aos elementos de S tem que ser igual a $c(S)$.

Referente à condição de folga do dual, sabemos que no final vamos encontrar uma solução exata envolvendo os valores x_S, ou seja, no final todo x_S vai ter valor 0 ou 1, então com certeza cada elemento pode ser coberto

até f vezes por f conjuntos diferentes.

Apresentadas essas ideias, temos o Algoritmo 24 que é uma f-aproximação.

Algoritmo 24: COBERTURA-PRIMALDUAL(E, $\mathcal{S}$, c)

Entrada: Conjunto E, coleção de conjuntos finitos $\mathcal{S}$ e função de custo nos conjuntos de $\mathcal{S}$

Saída: Cobertura x de E

```
1 Considere 0 representação do vetor nulo
2 x = 0
3 y = 0
4 enquanto existe elemento em E ainda não coberto faça
5 |   Tome um elemento e ainda não coberto e aumente y_e até que
  |    algum conjunto S esteja justo
6 |   Adicione todos os conjuntos justos, no momento, à solução x
7 |   Declare todos os elementos que estejam em x como cobertos
8 fim
9 retorna x
```

O Algoritmo 24 inicia, como já citamos, com valores de solução triviais para o primal e o dual. Enquanto não obtivermos uma solução viável para o primal, o algoritmo vai alterando os valores de y_e, ou seja, do dual, até que obtenhamos uma solução viável. Nessa relação entre primal e dual que temos a aplicação do esquema primal-dual. Veja que o algoritmo é polinomial pois o laço é $O(|E|)$, sendo que os processos dentro desse laço consumem $O(|E| + |\mathcal{S}|)$, logo o algoritmo é polinomial.

Segue o teorema que enuncia o fator de aproximação desse algoritmo.

Teorema 6.3. *Dada uma instância $\langle E, \mathcal{S}, c\rangle$ para a Cobertura por Conjuntos, o Algoritmo 24 é uma f-aproximação para esse problema.*

Demonstração. Da maneira que o algoritmo funciona é impossível que algum elemento seja deixado como não coberto, logo temos como resposta sempre uma solução viável para o problema. Além disso, veja que as restrições do dual também são sempre atendidas, já que nunca iremos aumentar um y_e

caso pelo menos um conjunto esteja justo. Dessa forma, a solução do primal e a solução do dual são viáveis.

Como $\alpha = 1$ e $\beta = f$, pela Proposição 6.1 sabemos que

$$\sum_{S \in \mathcal{S}} x_S . c(S) \leq f . \sum_{S \in \mathcal{S}} \sum_{e \in S} y_e$$

mas, como já desenvolvemos anteriormente nessa seção, podemos chegar à seguinte relação

$$\sum_{S \in \mathcal{S}} x_S . c(S) \leq f . OPT_{CC}(E, \mathcal{S}, c).$$

□

Considerações Finais

Nessa seção, a última desse relatório, apresentaremos nossas considerações finais sobre o trabalho. Começaremos com uma revisão dos objetivos gerais e específicos estabelecidos no início do projeto, depois apresentaremos como cada um desses objetivos foi trabalhado ao longo do projeto. Com essa estrutura, esperamos expor as considerações do aluno beneficiário sobre cada parte do projeto e como o projeto influenciou em sua formação.

Vamos começar relembrando o objetivo geral do projeto, que era introduzir o aluno à pesquisa na área de otimização combinatória através do estudo de algoritmos de aproximação e suas técnicas. Devido a esse objetivo, nessa seção, tentaremos a todo momento citar como cada parte do projeto contribui para que esse objetivo fosse atingido.

Na proposta do projeto foi definido um cronograma com diversos objetivos específicos relacionados ao assunto principal do projeto, que são os algoritmos de aproximação. Segue a lista de objetivos específicos estabelecidos:

1. Revisão de conceitos de projeto de análise de algoritmos e alguns conceitos em teoria dos grafos;
2. Estudo dos conceitos de redução entre problemas e complexidade (classes $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-completo e $\mathcal{NP}$-difícil);
3. Introdução a algoritmos de aproximação e algoritmos básicos (cobertura por vértices, escalonamento, caixeiro viajante e árvore de Steiner);
4. Estudo de algoritmos que utilizam redução entre problemas (cobertura por conjuntos e supercadeia mínima);

5. Estudo de algoritmos em grafos (corte multiseparador, k-corte e k-centro);
6. Estudo de esquemas de aproximação (mochila e empacotamento);
7. Revisão de conceitos de programação linear;
8. Estudo do problema da Cobertura por Conjuntos e seus algoritmos com as técnicas de arredondamento, *dual-fitting* e primal-dual;
9. Estudo da inaproximabilidade.

Perceba que essa lista de objetivos pode ser vista como uma lista de assuntos que o beneficiário deveria estudar. Cada um dos assuntos listados foi estudado de alguma forma pelo aluno ao longo do projeto. Todos os assuntos foram trabalhados seguindo um mesmo roteiro:

1. O beneficiário estudou o assunto através de livros e artigos selecionados pela orientadora;
2. O beneficiário e a orientadora se reuniram para discutir o assunto e sanar dúvidas;
3. O beneficiário escreveu no relatório sobre o assunto discutido em reunião;
4. A orientadora notificou o beneficiário de possíveis melhorias no relatório;
5. O beneficiário, após compreender o porquê das melhorias, corrigiu o relatório.

Todo esse processo contribuiu com o aprendizado do aluno, pois permitiu que o conteúdo não fosse somente estudado de maneira passiva. Veja que houve a discussão de ideias em reuniões com a orientadora e o exercício da escrita por meio desse relatório. Essas etapas permitiram que o aluno desenvolvesse sua escrita acadêmica e a capacidade de lidar com conceitos matemáticos.

A formação do aluno também foi beneficiada pelo projeto, pois permitiu que ele entrasse em contato com assuntos que não seriam normalmente vistos em sua graduação em ciência da computação.

No Capítulo 1 foram apresentados os assuntos referentes aos Objetivos 1 e 2, conceitos básicos de otimização combinatória, necessários para começar a estudar os problemas de otimização e o algoritmos de aproximação para os mesmos. Vale destacar que o aluno também estudou esses assuntos nas disciplinas de Análise de Algoritmos e Teoria dos Grafos, cursadas em paralelo ao projeto.

O Capítulo 2 cobriu os Objetivos 3, 4 e 5, permitindo que o aluno entrasse em contato com uma vasta gama de problemas combinatórios e conceitos matemáticos, além de algoritmos e técnicas de demonstração. Um assunto que teve muito destaque nesses capítulos foram os grafos, isso porque vários dos problemas estudados nesses capítulos envolvem grafos em sua definição.

No Capítulo 3 foi definido o que é um esquema de aproximação e foram apresentados alguns exemplos de esquemas de aproximação, ou seja, foi onde o Objetivo 6 foi trabalhado. Isso permitiu ao beneficiário ampliar sua visão sobre algoritmos de aproximação no geral, especialmente porque os fatores de aproximação dos esquemas de aproximação são uma generalização que cobre os fatores de aproximação dos algoritmos estudados no Capítulo 2.

No Capítulo 4 foram apresentados conceitos fundamentais de programação linear, assunto que refere-se ao Objetivo 7. Como o aluno ainda não tinha tido contato com a programação linear, a escrita desse capítulo também serviu como uma oportunidade para o aluno entender a importância da programação linear no projeto de algoritmos de maneira geral.

Os assuntos e conceitos apresentados no Capítulo 4 foram aplicados nos capítulos seguintes. Em especial, o Capítulo 6, que trata do Objetivo 8, teve como foco aplicar técnicas que envolvem programação linear a um problema de otimização combinatória já conhecido, no caso, a Cobertura por Conjuntos. No Capítulo 6, o aluno teve a possibilidade de ver como a programação linear e seus conceitos, como dualidade e relaxação, podem ser aplicados no projeto e análise de algoritmos.

Embora o tópico de inaproximabilidade, Objetivo 9, não tenha sido tra-

tado em um capítulo ou seção específica do relatório, foram vistos ao longo do projeto resultados relacionados à inaproximabilidade que permitiram contato do beneficiário com o assunto. Por exemplo, na Seção 2.4 mostramos que o problema do Caixeiro Viajante é inaproximável a não ser que $\mathcal{P} \neq \mathcal{NP}$. Com esse resultado e sua demonstração já foi possível entender o que faz um problema ser inaproximável e a importância das reduções nas demonstrações de resultados desse tipo.

Então, podemos ver que cada um dos objetivos específicos introduziu, de alguma forma, para o beneficiário um tópico diferente de otimização combinatória.

Algo destacável é que o estudo de algoritmos probabilísticos não foi inicialmente proposto no projeto. Porém, devido a um desejo do próprio aluno em estudar o tema, visando complementar o projeto, o tópico foi adicionado à lista de assuntos a serem estudados resultando no Capítulo 5. Esse capítulo possibilitou tanto o estudo de algoritmos probabilísticos aproximativos quanto de algoritmos probabilísticos em geral.

Agora, voltando nossas atenções ao objetivo geral do projeto. O presente relatório técnico, que foi escrito pelo aluno, teve papel fundamental para que o objetivo geral fosse atingido, pois sua redação exigiu que o beneficiário não somente estudasse os conteúdos, mas também discutisse ideias e as organizasse suficientemente bem a ponto de redigir um texto claro sobre o assunto.

Pelas razões apresentadas nessa seção, podemos dizer que esse projeto de iniciação científica cumpriu seu papel e realmente iniciou o aluno à área de otimização combinatória, enquanto ao mesmo tempo contribuiu no desenvolvimento de diversas habilidades como a escrita acadêmica, a compreensão de conceitos matemáticos e a exposição de ideias de maneira clara.

Referências Bibliográficas

[1] J. A. Bondy and U. S. R. Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1846289696.

[2] M. H. Carvalho, M. R. Cerioli, R. Dahab, P. Feofiloff, C. G. Fernandes, F. C. E., K. S. Guimarães, F. K. Miyazawa, J. C. Pina Jr., J. A. R. Soares, and Y. Wakabayashi. *Uma introdução sucinta a algoritmos de aproximação*. 2001.

[3] T. H. Cormen, C. E. Leiserson, R. Rivest, and S. C. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[4] F. K. Miyazawa. Programação inteira.

[5] L. Trevisan. Inapproximability of combinatorial optimization problems. *Paradigms of Combinatorial Optimization: Problems and New Approaches*, 2, 10 2004. doi: 10.1002/9781118600207.ch13.

[6] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2nd edition, 2003. ISBN 978-3-662-04565-7.

[7] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.